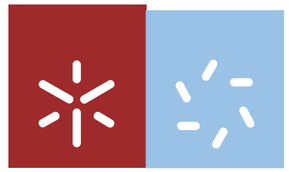
Isabel Maria Ventura Santos

Lógica da Separação e Verificação Formal de Programas

UMinho | 2012

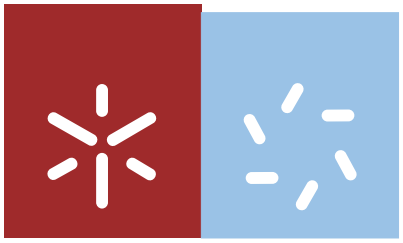


Universidade do Minho
Escola de Ciências

Isabel Maria Ventura Santos

Lógica da Separação e Verificação Formal de Programas

Outubro de 2012



Universidade do Minho
Escola de Ciências

Isabel Maria Ventura Santos

Lógica da Separação e Verificação Formal de Programas

Dissertação de Mestrado
Mestrado em Matemática e Computação

Trabalho realizado sob a orientação do
Professor Doutor Luís Filipe Ribeiro Pinto
e da
Professora Doutora Maria João Frade

Outubro de 2012

Agradecimentos

A elaboração desta dissertação não teria sido possível sem a ajuda de diversas pessoas às quais gostaria de agradecer por toda a disponibilidade.

Aos meus orientadores, Professora Maria João Frade e ao Professor Luís Pinto por todo o trabalho de supervisão e disponibilidade prestada sem os quais este trabalho não seria possível. Agradeço ainda as inúmeras revisões a este trabalho, bem como as palavras de encorajamento.

Aos meus amigos e colegas (Licenciatura em Matemática, Ciências da Computação, Engenharia Informática, Mestrado em Matemática e Computação e do Mestrado em Engenharia Informática) que me acompanharam e apoiaram ao longo de todo este processo. Em especial gostaria de agradecer aos "Nunos", aos "Pedros" e ao Zé pelas inúmeras discussões de informática à hora do almoço.

Um agradecimento muito especial ao Rui, pela motivação, apoio, suporte e inúmeros conhecimentos transmitidos sem o qual esta tarefa teria sido muito mais complicada.

Por fim mas não menos importante à minha família por toda a compreensão e paciência para mim, em especial aos meus pais e irmão.

Este trabalho é financiado por Fundos Nacionais através da FCT - Fundação para a Ciência e a Tecnologia no âmbito do projeto PTDC/EIA-CCO/105034/2008.

This work is funded by National Funds through the FCT - Fundação para a Ciência e a Tecnologia (Portuguese Foundation for Science and Technology) within project PTDC/EIA-CCO/105034/2008.

Resumo

A lógica da separação é uma lógica de programas talhada para a verificação dedutiva de programas imperativos que lidam com memória dinâmica. Uma das técnicas mais populares na verificação dedutiva de programas é usar a semântica axiomática para gerar condições de verificação, que são depois enviadas para ferramentas de prova. A componente responsável por gerar estas condições, para um dado triplo de Hoare, denomina-se *gerador de condições de verificação* (ou VCGen).

Esta dissertação apresenta um estudo sobre verificação dedutiva de programas imperativos com memória dinâmica, baseado na lógica da separação, onde é desenvolvido um VCGen para esta lógica de programas e é provada a correcção do VCGen.

No estudo, introduz-se uma linguagem imperativa simples com memória dinâmica, a linguagem $While_{pr}$, apresentando a sua sintaxe e a sua semântica operacional. Apresenta-se uma variante adequada da lógica da separação para esta linguagem (sintaxe, semântica e um sistema dedutivo base–o sistema **S**), sendo provada a correcção do sistema deductivo face à semântica dos triplos de Hoare.

Para encontrar um algoritmo de VCGen devidamente fundamentado, foram desenvolvidos sistemas dedutivos intermédios, mais adequados à mecanização do processo de prova, nomeadamente, o sistema $\mathbf{S^g}$ e o sistema $\mathbf{S^{gb}}$ (que é já um sistema *goal directed*), e foi provada a equivalência destes sistemas formais.

O VCGen obtido resulta da escolha de uma estratégia de prova no sistema $\mathbf{S^{gb}}$ baseada no cálculo da *weakest precondition*. É demonstrada a correcção deste VCGen face ao sistema dedutivo $\mathbf{S^{gb}}$, e, consequentemente, face ao sistema original **S** e face à semântica.

Foi construído um protótipo na linguagem Haskell que implementa o algoritmo de VCGen desenvolvido.

Abstract

Separation logic is an appropriate program logic for deductive verification of imperative programs which manipulate heap. One of the most popular approaches for deductive verification of programs is the use of axiomatic semantics for generation of the verification conditions, which are then passed to a proof tool for validation. The component of the verification arquitechture that generates the verification conditions relative to an Hoare triple is called a VCGen (*verification conditions generator*).

This dissertation presents a study on deductive verification of imperative programs with heap manipulation, based on separation logic, in which a VCGen for the underlying program logic was defined and the correctness of the VCGen was proved.

An imperative language with heap manipulation called $While_{pr}$ was introduced, and its syntax and operational semantics were defined. A variant of separation logic adequate to this language was defined (syntax, semantics and a deductive system –called system **S**), and soundness of the deductive system w.r.t. the semantics was proved.

In order to develop a correct VCGen, two intermediate deductive systems, more adequate for mechanization, were designed: system $\mathbf{S^g}$ and system $\mathbf{S^{gb}}$ (where the latter is already goal directed). Equivalence of all systems was established. The VCGen obtained, derived from system $\mathbf{S^{gb}}$, implements a proof strategy based on the calculation of *weakest preconditions*. Correctness of the VCGen w.r.t. $\mathbf{S^{gb}}$ (hence also w.r.t. the base system **S**, and to the semantics) was proved.

A prototype for the VCGen developed in the study was implemented in Haskell.

Conteúdo

Lista de Figuras

Capítulo 1

Introdução

Esta dissertação apresenta um estudo sobre verificação dedutiva de programas imperativos com memória dinâmica, baseado na lógica da separação. Esta lógica permite estabelecer a correção de um programa face a uma especificação previamente estipulada. No trabalho desenvolvido, em particular, foi desenhado um algoritmo capaz de gerar um conjunto de fórmulas lógicas que caraterizam a correção de um programa e foi provada a correção deste algoritmo.

1.1 Motivação

Os sistemas informáticos estão presentes, de modo direto ou indireto, na maioria das tarefas do nosso dia a dia. Os constantes avanços tecnológicos requerem software cada vez mais complexo. Cada vez mais confiamos no funcionamento adequado do hardware e dos componentes de software que o comanda, mas a complexidade dos componentes de software construídos leva a que mais facilmente se comentam erros de programação. O mau funcionamento de um componente de software é evidentemente algo a evitar. Mas enquanto em certas situações isso apenas causa alguns incómodos, outras há em que a ocorrência de erros não é de todo tolerável, dado o enorme impacto, de consequências muitas vezes irreversíveis, que estes erros podem causar. É o caso dos chamados sistemas críticos, de que são exemplo os sistemas ligados à aviação (e sistemas de transportes em geral), centrais nucleares, sistemas de tratamento e diagnóstico médico, etc. Mesmo em sistemas não críticos, a ocorrência de erros em software é sempre negativa para a empresa que o produziu e pode acarretar consequências financeiras negativas.

Por todas estas razões, as técnicas de validação de software assumem um papel cada vez mais importante no ciclo de desenvolvimento de software.

A técnica de validação de software mais usada na prática é o *teste*. O teste é uma técnica de validação em que o software é executado com uma coleção de inputs pré-definida (os testes), sendo depois confrontados os resultados obtidos com os resultados esperados. Existem ferramentas para auxiliar o teste, por exemplo gerando casos de teste com uma grande cobertura (número de traços do código testado) que,

quando bem utilizadas, permitem aumentar os níveis de confiança no código. Contudo, o teste exaustivo do código é, na maioria dos casos, impossível de alcançar. O teste não é portanto uma técnica completa: pode revelar a presença de erros, mas não consegue garantir a sua ausência.

A crescente complexidade do software, e as crescentes exigências no que concerne à sua correção e segurança, conduziram ao desenvolvimento dos métodos formais na engenharia de software. Esses métodos aplicam técnicas matemáticas na especificação, desenvolvimento e verificação de programas, e há cada vez mais ferramentas que auxiliam nestas tarefas. Aliás, a mecanização e automatização (dentro do possível) destes processos é essencial à aplicação de métodos formais no desenvolvimento de software, dada a sua complexidade e dimensão.

A noção de *especificação* é central aos métodos formais. Uma especificação é um modelo de um sistema de computação que descreve o comportamento desejado para esse sistema. Ou seja, *o que* queremos implementar, por oposição a *como* o vamos fazer. Coloca-se então o problema de garantir que uma implementação tem realmente o comportamento que é especificado, isto é, que a implementação é correta face à especificação dada. Existem duas linhas de investigação na abordagem a este problema:

- Derivar a implementação a partir da especificação, baseada em métodos formais que garantem que a implementação seja correta, i.e., garantir a correção *por construção*.

- Apresentar uma prova formal da correção da implementação face à especificação, i.e., garantir a correção *por verificação* formal.

Dentro da área da verificação formal existem duas noções que se destacam: o conceito de *sistema de transição de estados*, que permite capturar a essência operacional do sistema de computação que está a ser modelado; e o conceito de *lógica de programas*, que permite capturar a essência comportamental de um programa.

A verificação de modelos (*model checking*) é uma técnica de verificação centrada na demonstração de propriedades de um modelo representado por um sistema de transição de estados, que surgiu nos anos oitenta. Tipicamente, a propriedade do modelo que se quer demonstrar é expressa por uma fórmula numa lógica temporal, e a verificação é feita através de um algoritmo de execução simbólica que atravessa o modelo, verificando que todas as configurações possíveis satisfazem essa propriedade. Quando o espaço de estados do modelo é finito, o algoritmo de *model checking* pode, em teoria, demonstrar a propriedade em causa. Se a propriedade não for válida é encontrado um contra-exemplo. Esta técnica sofre do problema de explosão de estados. O grafo de transição de estados cresce exponencialmente em relação ao tamanho do sistema, o que faz com que a cabal verificação do modelo possa ser impraticável. Ainda assim as ferramentas de *model checking* são das mais usadas na prática, dado o seu alto grau de automação. Embora essas ferramentas possam não ter a capacidade de garantir a propriedade que queremos analisar, são muito úteis para encontrar

problemas (i.e. contra-exemplos) nos modelos, e deste modo, permitem aumentar a confiança no sistema que se está a implementar.

A verificação de programas é uma técnica focada na verificação dedutiva de código, tendo por base uma determinada lógica de programas. A ideia é anotar os programas com condições lógicas que caraterizam o estado de execução do programa, e demonstrar formalmente que se executarmos um comando/programa a partir de um estado que valida determinadas condições (sintetizadas numa anotação, a que se chama *pré-condição*), conseguimos assegurar que o estado de chegada tem as características desejadas (sintetizadas numa anotação, a que se chama *pós-condição*). Esta linha de investigação surgiu nos anos sessenta, e deu origem à chamada semântica axiomática das linguagem de programação. Em termos práticos, esta abordagem está diretamente relacionada com a metodologia de desenvolvimento de software a que se dá o nome de *design-by-contract*, onde cada procedimento do código é anotado com uma pré e uma pós condição–o seu *contrato*– que deve ser assegurado.

Atualmente, várias das principais linguagens de programação integram uma camada de contratos. Podemos, por exemplo, citar as linguagens SPEC# (próximo do C#) e SPARK (um subconjunto do ADA) que suportam de forma nativa este paradigma do *design-by-contract* (originalmente introduzido pela linguagem Eiffel); ou ainda o ESC/Java e Krakatoa, que aplicam ao Java uma camada de anotações escrita em JML, ou o Frama-C que é uma plataforma de validação e análise de código C, que usa a linguagem de anotações ACSL. A prova de correção de um programa face ao seu contrato, é contudo um processo bastante exigente (em termos de dificuldade técnica e da especialização das pessoas envolvidas no processo) e só costuma ser posto em prática no desenvolvimento de software para sistemas críticos.

1.2 Verificação dedutiva de programas

A verificação dedutiva de programas baseia-se em lógicas de programas, cujo exemplo seminal é a lógica de Hoare [14, 13]. A lógica de Hoare foi introduzida por C.A.R. Hoare no final da década de sessenta, como uma lógica para raciocinar acerca de programas imperativos. Esta lógica lida com a noção de triplo de Hoare, $\{\phi\}C\{\psi\}$, onde ϕ e ψ são fórmulas em lógica de primeira ordem (a pré e a pós condição, respetivamente) e C é um comando/programa. A validade do triplo indica que se o comando C for executado a partir de um estado onde a pré-condição ϕ é válida, então, se a execução terminar, o estado a que se chega valida a pós-condição ψ. A ideia é que o comportamento de cada comando pode ser descrito por um sistema de inferência para os triplos de Hoare, que estabelece uma semântica axiomática para a linguagem de comandos em causa. A correção de um programa C face a um contrato (dado pela pré-condição ϕ e pós-condição ψ) é assegurada construindo uma derivação para o triplo $\{\phi\}C\{\psi\}$ nesse sistema de inferência.

Os triplo de Hoare que descrevemos acima referem-se à *correção parcial* (i.e. sem garantias de terminação) dos programas. Existe uma outra versão da noção de triplo de Hoare, em cuja interpretação se assegura a terminação do programa. Neste

caso costuma falar-se de *correção total* do programa.

Uma das técnicas mais convenientes de organizar um sistema de verificação de programas, é usar a semântica axiomática para gerar obrigações de prova (chamadas *condições de verificação*) que depois são enviadas para uma ferramenta de prova (automática ou assistida). A ideia é que se todas as condições de verificação forem válidas então o programa é garantidamente correto. O componente que recebe um triplo de Hoare e gera as condições de verificação necessárias e suficientes à sua demostração é chamado de *Gerador de Condições de Verificação* (ou *VCGen*).

A completa automatização do processo de verificação de um programa, é algo que é impossível de alcançar (desde logo, porque a lógica de primeira ordem não é decidível). A intervenção humana neste processo surge a dois níveis: anotando o código (por exemplo, escrevendo invariantes de ciclo) de forma a que a prova das condições de verificação seja mais simples; e intervindo ao nível da prova das condições que não conseguem ser demonstradas automaticamente.

A geração automática de invariantes de ciclo é uma área de investigação ativa e algumas plataformas de verificação de programas incorporam já alguma funcionalidade dentro deste contexto. Muito relevante, é o enorme avanço nos sistemas de prova automática (os chamados *theorem provers*) que se verificou na última década, com particular relevo para os *Satisfability Module Theories (SMT) solvers*, de que são exemplo o Z3, Yices, Alt-Ergo, CVC3 ou o Simplify.

Uma das vantagens das plataformas de verificação de programas baseadas em VCGens é a sua flexibilidade. As condições de verificação geradas podem ser enviadas para diferentes *solvers*, o que é uma vantagem, dado que diferentes *solvers* podem ter diferentes capacidades de prova, é mais provável que mais condições de verificação sejam descartadas automaticamente.

Note-se, mais uma vez, que é impossível automatizar completamente o processo de prova para teorias de primeira ordem. A resposta dos *SMT solvers* quando questionados sobre a validade de uma fórmula (condição de verificação) pode ser uma de três: a fórmula é válida, a fórmula não é válida (e é indicado um contra-exemplo, que ilustra uma situação em que a fórmula é falsa), ou *time-out* (porque não está a conseguir demonstrar que é válida, nem que não é). Neste último caso, será necessário que a prova seja efetuada manualmente, normalmente com o auxilio de um sistema de prova assistida, como por exemplo o Coq, Isabelle, PVS, HOL.

1.3 Lógica da separação

A maioria das linguagens de programação imperativas trabalham com memória dinâmica e lidam com apontadores (endereços de memória). Esta característica aumenta o poder expressivo da linguagem, e dá a possibilidade de implementar estruturas de dados dinâmicas. Contudo existem dificuldades inerentes à verificação de programas com memória dinâmica. Por exemplo, a presença de apontadores traz a possibilidade de, num programa, existirem múltiplas referências para uma mesma célula de

memória (fenómeno vulgarmente conhecido por *aliasing*). Este fenómeno dificulta o raciocínio sobre as propriedades do programa, uma vez que a alteração de uma célula de memória poderá ter reflexos em diferentes partes do programa.

Os apontadores são mecanismos de partilha de memória muito poderosos e flexíveis, úteis na implementação de estruturas de dados ligadas. No entanto, uma programação menos cuidadosa, com este tipo de estruturas, facilmente resulta em erros de programação difíceis de detetar. A verificação formal de programas que manipulam memória dinâmica (a *heap*) é assim um tópico de investigação muito ativo, e é neste contexto que esta dissertação se enquadra.

Para dar semântica às linguagens de programação que manipulam a *heap*, lidando com o problema do *aliasing*, e estudar as propriedades dos seus programas, a lógica de Hoare não se revela muito adequada. O problema deve-se ao facto de que para captar as restrições de partilha de memória nas estruturas de dados ligadas com que os programas lidam, são necessárias escrever condições em lógica de primeira ordem que são muito complexas, o que torna impraticável o processo de prova.

Para capturar as características deste tipo de linguagens e de programas é necessário definir um modelo de memória adequado. Foram propostas várias abordagens para resolver este problema, das quais se destaca o trabalho de Burstall [9] e os trabalhos subsequentes de Bornat [8], e de Reynolds e O'Hearn [19, 15, 22, 21, 23] que deram origem à lógica da separação.

A lógica da separação é uma extensão à lógica de Hoare para raciocinar acerca de programas imperativos que manipulam a *heap*. Esta lógica de programas assenta na lógica BI (das *bunched implications*) [20]. O modelo de memória, que constituí o estado dos programas, é composto por duas partes: a *store* que representa a memória estática do programa, e a *heap* que representa o espaço de memória dinâmica (que pode ser alocada e libertada ao longo da execução do programa).

Esta lógica põe em prática o princípio do "raciocínio local", que sintetiza a ideia de que para perceber como um programa funciona, deve ser possível confinar o raciocínio às células de memória que são efectivamente acedidas pelo programa; o valor de qualquer outra célula mantém-se inalterado. O raciocínio local é suportado pela introdução de novas conectivas lógicas, das quais se destaca a *conjunção separada*, $\phi * \psi$ que garante que as fórmulas ϕ e ψ são válidas em partes disjuntas da *heap*.

O poder expressivo destas novas conectivas e predicados que caraterizam a *heap*, e a introdução de regras de inferência no sistema formal que exploram esta característica de raciocínio local, permite raciocinar acerca dos programa de um modo mais modelar. De realçar a regra *frame* que estende uma propriedade demonstrada localmente para um nível global (que engloba o resto da *heap* que não foi alterada pelo programa).

A semântica dos triplos de Hoare na lógica da separação difere da da lógica de Hoare no que toca à questão de *safety*. Mais concretamente a validade de um triplo $\{\phi\}C\{\psi\}$ significa se o programa C executar a partir de uma estado inicial que satisfaça ϕ, então a sua execução é segura (i.e. não aborta) e, se o programa terminar, o seu estado final satisfaz ψ.

Esta semântica dos comandos faz com que os programas que acedem a posições de memória não alocadas executem para um estado de erro. Portanto, a validade de um triplo de $\{\phi\}C\{\psi\}$ assegura que a pré-condição ϕ descreve toda a memória (expecto a que é alocada por C) que pode ser acedida durante a execução de C (o *footprint* de C).

A semântica *safety* dos triplos e o raciocínio local sintetizado pela regra de *frame*, faz com que as provas em lógica da separação se desenvolvam de forma mais simples.

Existem várias ferramentas baseadas em lógica da separação, por exemplo, o Smallfoot [5, 7, 6] que faz verificação por execução simbólica do código; o Ynot [18, 11] que é uma biblioteca Coq [2] para lógica da separação com táticas de prova associadas; o HOLfoot [1, 25] que é uma implementação do Smallfoot para HOL4 que lida com diferentes tipos e dados e permite fazer prova interativa; ou o jStar [12] que é uma ferramenta de prova automática para Java.

1.4 Resumo da contribuição

Contribuições preliminares desta dissertação foram:

1. fixar a linguagem $While_{pr}$, a par de uma semântica operacional "small-step" para a linguagem, detalhando alguns aspectos que, por vezes, os trabalhos na área não especificam.

2. estabelecer uma apresentação da lógica da separação para a linguagem $While_{pr}$ e fixar um sistema dedutivo base para os triplos da lógica, provando a correção do sistema dedutivo em relação à semântica da lógica da separação.

Estas contribuições, embora preliminares e sem trazerem aspectos de inovação essenciais, eram indispensáveis para poder abordar-se a correção do processo de geração das condições de verificação com garantias de correção. A contribuição principal desta dissertação foi o desenvolvimento do algoritmo de *VCGen* para a linguagem $While_{pr}$ e a prova de que, de facto, este algoritmo é correto em relação à semântica da lógica da separação e em relação ao sistema dedutivo base.

Esta contribuição baseou-se na definição de sistemas dedutivos intermédios para os triplos da lógica (mais talhados para a mecanização do processo de geração de condições de verificação) e nas provas de equivalência da dedutibilidade nos três sistemas considerados.

Uma contribuição adicional desta dissertação foi a de desenvolver um protótipo de uma ferramenta que implementa o algoritmo de *VCGen* desenvolvido. O algoritmo implementado, dados uma pré-condição, um comando e uma pós-condição, descritos na representação Haskell estabelecida para asserções e comandos, produz o conjunto de asserções correspondente às condições de verificação exigidas à correção do programa.

1.5 Estrutura da dissertação

Esta dissertação está organizada em seis capítulos e contém ainda dois anexos. O primeiro capítulo é introdutório, motivando o estudo efetuado e contextualizando a abordagem seguida no trabalho desenvolvido. Neste capítulo são também sumariadas as contribuições principais desta dissertação.

No segundo capítulo é apresentada a linguagem objeto do estudo, a linguagem $While_{pr}$, sendo definidas a sua sintaxe e a sua semântica, sendo esta última dada por uma semântica operacional *small-step*. Este capítulo inclui diversos exemplos que ilustram a linguagem.

No terceiro capítulo é apresentada a lógica da separação associada à linguagem $While_{pr}$. Em particular, é definida a semântica de triplos da lógica, é apresentado um sistema de inferência base para estes triplos e é provada a correção do sistema de inferência em relação à semântica. Este capítulo conclui com uma secção onde é definida uma teoria base para listas ligadas e são dados alguns exemplos de deduções que provam a validade de certos triplos envolvendo listas.

No quarto capítulo são estudados dois sistemas de inferência alternativos para a linguagem $While_{pr}$, mais talhados para o processo de verificação automática de programas. Ambos os sistemas apresentam regras globais (dispensando as regras de *frame* e de eliminação de variáveis auxiliares) e um dos sistemas (o sistema *goal-directed*), adicionalmente, dispensa ainda a regra da consequência. O algoritmo $VCGen$ para a geração de condições de verificação de um programa é então obtido a partir do sistema *goal directed*, sendo provada a correção e completude do $VCGen$ em relação a este sistema.

A implementação na linguagem Haskell que foi efetuada do algoritmo de $VCGen$ é apresentada no quinto capítulo. Em particular, são descritas as estruturas usadas para representar os vários ingredientes da lógica da separação. O $VCGen$ implementado é ilustrado com alguns exemplos já considerados em capítulos anteriores.

Por fim, no sexto capítulo, são apresentadas conclusões do trabalho desenvolvido e são também apontadas algumas possibilidades de temas de interesse para trabalho futuro.

Capítulo 2

$While_{pr}$: uma linguagem *While* com apontadores

Neste capítulo vamos apresentar a linguagem de programação $While_{pr}$, que será objeto do nosso estudo ao longo desta dissertação. A linguagem $While_{pr}$ é um linguagem imperativa muito simples que permite a manipulação de memória dinâmica. Basicamente, é a linguagem *While*, originalmente apresentada por Hoare [14, 13], estendida com operações de manipulação de memória dinâmica. Assim, para além dos comandos *skip*, atribuição, sequenciação, ciclo *while* e condicional, teremos ainda comandos para alocação, libertação, leitura e escrita em memória dinâmica.

Neste trabalho, ao contrário dos trabalhos [21, 23, 15, 19], optámos por distinguir os tipos de dados associados a inteiros e endereços. Esta distinção torna o estudo da linguagem "mais pesado", mas, ao mesmo tempo, permite evitar algumas potenciais ambiguidades sobre a linguagem $While_{pr}$.

Para simplificar as questões relacionadas com a tipificação, optámos por distinguir as variáveis do programa de tipo inteiro e de tipo endereço, e estipular que as células de memória alocadas na *heap* têm tamanho fixo, e são estruturas de campos etiquetados, sendo cada campo de tipo predefinido. A simplificação de ter uma *heap* de *records* de tamanho fixo (que pode ser também ser encontrada em [15]) permite-nos focar nos aspectos essenciais do tratamento formal da memória dinâmica através da lógica da separação.

2.1 Sintaxe

A linguagem tem três tipos básicos: *booleanos* (*Bool*), inteiros (*Int*) e endereços (*Loc*). As expressões *booleanas* são formadas a partir da constante *true* ($\top$), por comparação de expressões do mesmo tipo, e combinando expressões *booleanas* através de operadores *booleanos* (conjunção, disjunção e negação). As expressões de tipo *Int* são formadas a partir de constantes (valores inteiros), e de um conjunto de variáveis de tipo *Int*, juntamente com os operadores usuais (soma, multiplicação, divisão

inteira, etc). Usaremos $x, y, z, ...$ para representar as variáveis de tipo *Int*; $\mathbf{V}_{Int}$ denota o conjunto de todas estas variáveis.

Uma expressão do tipo *Loc* só têm duas formas possíveis: ou é a constante *nil* (que representa o apontador nulo) ou é uma variável. Usaremos $p, p', p_1, ...$ para representar as variáveis do tipo *Loc* e $\mathbf{V}_{Loc}$ denota o conjunto destas variáveis.

$\mathbf{V}ar = \mathbf{V}_{Loc} \cup \mathbf{V}_{Int}$ e utilizaremos $v, v', v_1, ...$ para representar variáveis do conjunto $\mathbf{V}ar$. Assumimos que o tipo da variável v é dado por $\mathbf{type}(v)$.

Os *records* alocados na *heap* são sempre do mesmo formato. $\mathbf{F}ields$ denota o conjunto (finito) dos nomes dos campos dos *records*, e usaremos $f, f_1, ...$ para representar esses nomes. Cada campo do *record* tem um tipo fixo que poderá ser *Int* ou *Loc*. Assumimos que o tipo associado a um campo f é dado por $\mathbf{type}(f)$. Os *records* estão presentes apenas na *heap* e só podem ser acedidos campo a campo.

$\mathbf{Exp}_{Int} \ni e ::= ... \mid -1 \mid 0 \mid 1 \mid ... \mid x \mid -e \mid e + e \mid e \times e \mid e \div e \mid e \bmod e$, com $x \in \mathbf{V}_{Int}$

$\mathbf{Exp}_{Loc} \ni l ::= nil \mid p$, com $p \in \mathbf{V}_{Loc}$

$\mathbf{Exp}_{Bool} \ni b ::= \top \mid \neg b \mid b \wedge b \mid b \vee b \mid e_1 = e_2 \mid e_1 \leq e_2 \mid e_1 \geq e_2 \mid e_1 < e_2 \mid e_1 > e_2 \mid l_1 = l_2$

Figura 2.1: Expressões da linguagem $While_{pr}$

Na Figura 2.1 apresenta-se a sintaxe abstrata das expressões da linguagem $While_{pr}$. Note-se que usamos e, l, b para representar as expressões do tipo *Int*, *Loc* e *Bool*, respetivamente. Escreveremos $a, a', ...$ para representar expressões do tipo *Int* ou *Loc*. Depois de definidas as várias expressões da linguagem, podemos finalmente definir os comandos da linguagem.

$\mathbf{C}omm \ni \mathsf{C}$	$::= \mathtt{skip}$		Skip
	$\mid v := a$	se $a \in \mathbf{Exp}_{\mathbf{type}(v)}$	Atrib.
	$\mid \mathsf{C}_1; \mathsf{C}_2$		Comp.
	$\mid \mathtt{if}\ b\ \mathtt{then}\ \mathsf{C}_t\ \mathtt{else}\ \mathsf{C}_f$		If
	$\mid \mathtt{while}\ b\ \{\mathsf{C}\}$		While
	$\mid p \rightarrow f := a$	se $a \in \mathbf{Exp}_{\mathbf{type}(f)}$	Mutation
	$\mid \mathtt{new}(p)$		Allocation
	$\mid \mathtt{dispose}(p)$		Dispose
	$\mid v := p \rightarrow f$	se $v \in \mathbf{V}_{\mathbf{type}(f)}$	Lookup

Figura 2.2: Classe dos comandos

A sintaxe abstrata dos comandos da linguagem $While_{pr}$ está definida na Fi-

gura 2.2. Como se pode ver, esta linguagem é uma extensão da linguagem *While* simples com novos comandos para a manipulação de memória dinâmica. Para além dos usuais comandos *skip*, atribuição, sequenciação, condicional, ciclo *while*, temos os comandos:

`new`(p) - que aloca um *record* na *heap* e coloca na variável p o endereço do *record* alocado.

`dispose`(p) - que liberta da *heap* a memória referente ao *record* que está no endereço guardado na variável p.

$p{\rightarrow}f{:=}a$ - que coloca no campo f do *record* apontado por p o valor da expressão a.

$v{:=}p \rightarrow f$ - que coloca na variável v o valor que está no campo f do *record* apontado por p (isto é, que está na *heap*, no endereço guardado em p).

Note que os comandos que manipulam a *heap* são apenas estes quatro. As expressões não dependem da *heap* e como veremos na secção seguinte, a avaliação das expressões necessitará apenas de "consultar" a *store* (o valor atribuído a cada variável). Na secção seguinte veremos também como se processa a avaliação dos comandos.

2.2 Semântica

A semântica dos programas é suportada pela noção de estado. Numa linguagem *While* simples, sem memória dinâmica, o estado é uma função de valoração das variáveis do programa. Para dar semântica aos programas da linguagem $While_{pr}$, temos de estender o estado de forma a capturar a noção de *heap* (onde a memória é alocada e libertada). Assim, o estado é constituído por duas partes: a *store*, onde se mapeia cada variável no seu valor, e a *heap*, onde se mapeiam os endereços (dos espaços alocados) no seu conteúdo. Como as células alocadas na *heap* são *records*, o valor de cada célula é modelado por uma função finita que para cada campo do *record* dá o seu valor.

Antes de apresentar a definição formal da noção de estado, introduziremos alguma notação.

Notação 2.2.1 Sejam A, B conjuntos tais que:

- $A + B$ representa a união disjunta de A e B
- $A \times B$ representa o produto cartesiano de A por B
- $A \rightarrow B$ representa uma função total de domínio A e conjunto de chegada B
- $A \hookrightarrow_{fin} B$ representa uma função finita parcial de domínio A e conjunto de chegada B
- escrevemos $[x_1 : a_1, ..., x_n : a_n]$ para representar a função finita de domínio $\{x_1, ..., x_n\}$ que mapeia cada x_i em a_i

- $[\,]$ denota a função vazia (isto é, cujo domínio é o conjunto vazio)

S$tate$, **S**$tore$ e **H**eap denotarão os conjuntos de todos os estados, *stores* e *heaps* respetivamente.

Definição 2.2.2 Um *estado* é um par $(s,h) \in \mathbf{S}tate$ onde

$$\begin{array}{l}\mathbf{S}tate = \mathbf{S}tore \times \mathbf{H}eap.\\ \mathbf{S}tore = \mathbf{V}ar \to \mathbf{V}alues,\\ \mathbf{V}alues = \mathbb{Z} + \mathbf{A}ddress + \{\mathbf{nil}\}\\ \mathbf{H}eap = \mathbf{A}ddress \hookrightarrow_{fin} \mathbf{R}ecords\\ \mathbf{R}ecords = \mathbf{F}ields \to \mathbf{V}alues\end{array}$$

O conjunto **A**$ddress$ tem que ser infinito. A necessidade de impor que **A**$ddress$ seja um conjunto infinito deve-se ao facto de querermos modelar a *heap* como um recurso que não se esgota. Como consequência, será sempre possível alocar memória na heap.

A interpretação das expressões é puramente funcional (isto é, não provocam alteração no estado) e toda a expressão sintaticamente válida tem sempre um valor qualquer que seja o estado em que é avaliada. Além disso, a interpretação das expressões só depende da *store* (uma vez que a manipulação da *heap* é feita com recurso a comandos específicos). Como era de esperar, expressões do tipo *Bool* são interpretadas pelos valores lógicos (T ou F), expressões do tipo *Int* por valores em $\mathbb{Z}$ e expressões do tipo *Loc* como endereços (elementos do conjunto $\mathbf{A}ddress + \{\mathbf{nil}\}$), sendo que a constante *nil* é interpretada como um endereço especial que corresponde ao apontador nulo.

Definição 2.2.3 (Semântica das expressões $While_{pr}$) $[\![.]\!]$ mapeia cada expressão de inteiros $e \in \mathbf{Exp}_{Int}$ numa função $[\![e]\!] : \mathbf{S}tore \to \mathbb{Z}$, cada expressão de *booleanos* $b \in \mathbf{Exp}_{Bool}$ numa função $[\![b]\!] : \mathbf{S}tore \to \{T,F\}$ e cada expressão de endereço $l \in \mathbf{Exp}_{Loc}$ numa função $[\![l]\!] : \mathbf{S}tore \to \mathbf{A}ddress + \{\mathbf{nil}\}$:

- $[\![e]\!] : \mathbf{S}tore \to \mathbb{Z}$ é definida recursivamente por:

$$\begin{array}{l}[\![z]\!](s) = z \quad (z \in \mathbb{Z})\\ [\![x]\!](s) = s(x)\\ [\![-e]\!](s) = -[\![e]\!](s)\\ [\![e_1 + e_2]\!](s) = [\![e_1]\!](s) + [\![e_2]\!](s)\\ [\![e_1 - e_2]\!](s) = [\![e_1]\!](s) - [\![e_2]\!](s)\\ [\![e_1 \times e_2]\!](s) = [\![e_1]\!](s) \times [\![e_2]\!](s)\\ [\![e_1 \div e_2]\!](s) = \begin{cases} [\![e_1]\!](s) \div [\![e_2]\!](s), & \text{se } [\![e_2]\!](s) \neq 0\\ 0, & \text{caso contrário}\end{cases}\\ [\![e_1 \bmod e_2]\!](s) = \begin{cases} [\![e_1]\!](s) \bmod [\![e_2]\!](s), & \text{se } [\![e_2]\!](s) \neq 0\\ 0, & \text{caso contrário}\end{cases}\end{array}$$

- $[\![l]\!] : \mathbf{S}tore \to \mathbf{A}ddress + \{\mathbf{nil}\}$ é definida por:
  $[\![nil]\!](s) = \mathbf{nil}$
  $[\![p]\!](s) = s(p)$

- $[\![b]\!] : \mathbf{S}tore \to \{T, F\}$ é definida recursivamente por:
  $[\![\top]\!](s) = T$
  $$[\![\neg b]\!](s) = \begin{cases} T, & \text{se } [\![b]\!](s) = F \\ F, & \text{caso contrário} \end{cases}$$
  $$[\![b_1 \wedge b_2]\!](s) = \begin{cases} F, & \text{se } [\![b_1]\!](s) = F \\ [\![b_2]\!](s), & \text{caso contrário.} \end{cases}$$
  $$[\![b_1 \vee b_2]\!](s) = \begin{cases} T, & \text{se } [\![b_1]\!](s) = T \\ [\![b_2]\!](s), & \text{caso contrário.} \end{cases}$$
  $[\![e_1 \odot e_2]\!](s) = [\![e_1]\!](s) \odot [\![e_2]\!](s)$ onde $\odot \in \{=, \neq, <, \leq, >, \geq\}$
  $[\![l_1 = l_2]\!](s) = [\![l_1]\!](s) = [\![l_2]\!](s)$

Vamos agora definir uma semântica operacional para os comandos. O comportamento de um comando é descrito por uma transformação do estado. Mais concretamente, o significado de um comando será dado por uma semântica operacional *small-step* (isto é, com ênfase nos passos individuais de execução do programa), baseada na definição de uma relação de transição, $\rightsquigarrow$, entre configurações. Uma configuração será um par (comando, estado) ou *abort*, ou seja, $\rightsquigarrow \subseteq (\mathbf{C}omm \times \mathbf{S}tate) \times ((\mathbf{C}omm \times \mathbf{S}tate) + \{abort\})$.

Antes de apresentarmos a relação de transição, precisamos de introduzir alguma notação sobre funções.

Notação 2.2.4 Seja g uma função de A para B, possivelmente parcial:

- $dom(g)$ denota o domínio da função g, isto é, os elementos de A para os quais a função está definida

- $g\rceil C$ denota a restrição da função g ao domínio C, isto é, $g\rceil C(x) = g(x)$ se $x \in C \cap dom(g)$

- $$[g|x : a](y) = \begin{cases} a, & \text{se } x = y \\ g(y), & \text{se } x \neq y \wedge y \in dom(g) \end{cases}$$

Note-se em particular que:

- $dom([g|x : a]) = dom(g) \cup \{x\}$

- $g = [\,] \Leftrightarrow dom(g) = \varnothing$

Definição 2.2.5 (Relação de transição entre configurações)

A relação $\rightsquigarrow \subseteq (\mathbf{C}omm \times \mathbf{S}tate) \times ((\mathbf{C}omm \times \mathbf{S}tate) + \{abort\})$ é definida indutivamente pelo conjunto de regras apresentadas nas Figuras 2.3 e 2.4.

Uma configuração $(\mathsf{C},(s,h))$ com $\mathsf{C} \neq \texttt{skip}$ diz-se *não terminal*. Uma configuração diz-se *terminal* se for da forma $(\texttt{skip},(s,h))$ ou *abort*.

Dadas configurações γ e γ' escrevemos $\gamma \rightsquigarrow^n \gamma'$ para indicar que existe uma sequência com n de transições de γ para γ' (isto é, existem $\gamma_1,...,\gamma_{n+1}$ tal que $\gamma = \gamma_1 \rightsquigarrow \gamma_2 \rightsquigarrow ... \rightsquigarrow \gamma_n \rightsquigarrow \gamma_{n+1} = \gamma'$), escrevemos $\gamma \rightsquigarrow^* \gamma'$ para indicar que $\gamma = \gamma'$ com $\gamma \rightsquigarrow^n \gamma'$ para algum $n \in \mathbb{N}_0$, e escrevemos $safe(\gamma)$ para indicar que $\gamma \not\rightsquigarrow^* abort$ (isto é, $\neg(\gamma \rightsquigarrow^* abort)$).

$$\frac{}{(v{:=}a,(s,h)) \rightsquigarrow (\texttt{skip},([s|v:[\![a]\!](s)],h))}\ \text{Atrib.}$$

$$\frac{(\mathsf{C}_1,(s,h)) \rightsquigarrow (\mathsf{C}_1',(s',h'))}{(\mathsf{C}_1;\mathsf{C}_2,(s,h)) \rightsquigarrow (\mathsf{C}_1';\mathsf{C}_2,(s',h'))}\ \text{Comp. 1}$$

$$\frac{}{(\texttt{skip};\mathsf{C}_2,(s,h)) \rightsquigarrow (\mathsf{C}_2,(s,h))}\ \text{Comp. 2}$$

$$\frac{[\![b]\!](s) = T}{(\texttt{if } b \texttt{ then } \mathsf{C}_t \texttt{ else } \mathsf{C}_f,(s,h)) \rightsquigarrow (\mathsf{C}_t,(s,h))}\ \text{If- True}$$

$$\frac{[\![b]\!](s) = F}{(\texttt{if } b \texttt{ then } \mathsf{C}_t \texttt{ else } \mathsf{C}_f,(s,h)) \rightsquigarrow (\mathsf{C}_f,(s,h))}\ \text{If- False}$$

$$\frac{[\![b]\!](s) = T}{(\texttt{while } b \{\mathsf{C}\},(s,h)) \rightsquigarrow (\mathsf{C};\texttt{while } b \{\mathsf{C}\},(s,h))}\ \text{While- True}$$

$$\frac{[\![b]\!](s) = F}{(\texttt{while } b \{\mathsf{C}\},(s,h)) \rightsquigarrow (\texttt{skip},(s,h))}\ \text{While- False}$$

Figura 2.3: Semântica dos comandos relativos à linguagem *While*

$$\frac{l \in \mathbf{A}ddress - dom(h)\ ,\ r \in \mathbf{R}ecords}{(\texttt{new}(p),(s,h)) \rightsquigarrow (\texttt{skip},([s|p:l],[h|l:r]))}\ \text{Allocation}$$

$$\frac{[\![p]\!](s) \in dom(h)}{(p{\rightarrow}f{:=}a,(s,h)) \rightsquigarrow (\texttt{skip},(s,[h|[\![p]\!](s):r]))}\ \text{Mutation}$$

$$\text{onde } r = [h([\![p]\!](s))|f:[\![a]\!](s)]$$

$$\frac{[\![p]\!](s) \notin dom(h)}{(p{\rightarrow}f{:=}a,(s,h)) \rightsquigarrow abort}\ \text{Mutation}$$

$$\frac{[\![p]\!](s) \in dom(h)}{(\texttt{dispose}(p),(s,h)) \rightsquigarrow (\texttt{skip},(s,h\restriction(dom(h)-\{[\![p]\!](s)\})))}\ \text{Dispose}$$

$$\frac{[\![p]\!](s) \notin dom(h)}{(\texttt{dispose}(p),(s,h)) \rightsquigarrow abort}\ \text{Dispose}$$

$$\frac{[\![p]\!](s) \in dom(h)}{(v{:=}p \rightarrow f,(s,h)) \rightsquigarrow (\texttt{skip},([s|v:(h([\![p]\!](s)))(f)],h))}\ \text{Lookup}$$

$$\frac{[\![p]\!](s) \notin dom(h)}{(v{:=}p \rightarrow f,(s,h)) \rightsquigarrow abort}\ \text{Lookup}$$

Figura 2.4: Semântica dos comandos que manipulam a heap

Nas Figuras 2.5, 2.6, 2.7 e 2.8 representa-se esquematicamente o efeito provocado pela execução de um comando. Ao longo das figuras mencionadas, assumiu-se que a *heap* está organizada em *records* de um único campo de nome *valor*, e cujo o tipo é *Int*. Portanto, nestes exemplos temos $\mathbf{F}ields = \{valor\}$

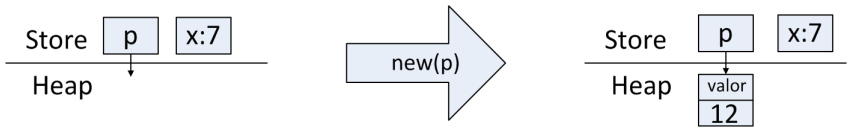


Figura 2.5: Execução do comando Allocation

Na Figura 2.5 podemos ver que partimos de uma *heap* vazia, sem quaisquer endereços alocados. Depois de executarmos o comando $\texttt{new}(p)$, a variável p perde o conteúdo anterior e passa a apontar para o novo endereço alocado na *heap*. Importa ainda salientar que não sabemos o conteúdo do campo valor, uma vez que o comando $\texttt{new}(p)$ aloca a memória na *heap* mas não faz a sua inicialização. Se o quisermos fazer teremos que usar, adicionalmente, o comando *mutation*.

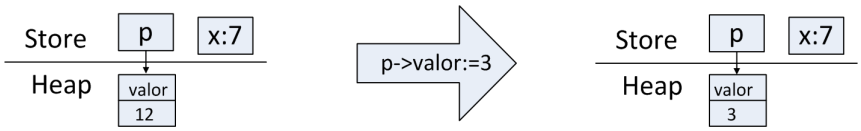


Figura 2.6: Execução do comando Mutation

Na Figura 2.6 podemos ver que na *heap* se encontra um endereço alocado. Nesse endereço, o campo *valor* tem o valor 12 e quando executarmos o comando $p \rightarrow valor := 3$ é alterado para 3.

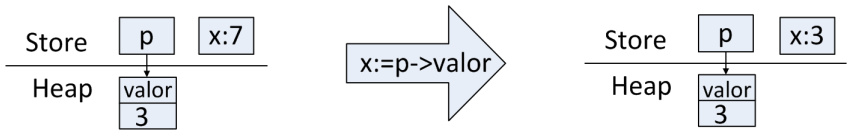


Figura 2.7: Execução do comando Lookup

Na Figura 2.7 a variável x tem inicialmente o valor 7, depois de executarmos o comando $x := p \rightarrow valor$, passa a ter o valor que se encontra no endereço p no campo *valor*, ou seja, o valor 3. Note que a *heap* não sofre alterações.

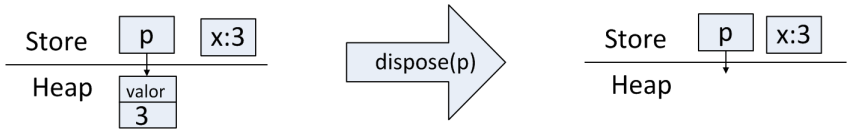


Figura 2.8: Execução do comando Dispose

Na Figura 2.8 partimos de uma *heap* com um endereço alocado (apontado por p). Depois de executarmos o comando $\texttt{dispose}(p)$ esse endereço deixa de estar alocado na *heap*. Repare que a *store* não sofre alterações. O apontador p continua a conter o mesmo endereço que tinha antes da execução do comando $\texttt{dispose}(p)$, só que agora nesse endereço não temos memória alocada.

Exemplo 2.2.6 Consideremos o seguinte programa que faz a troca entre os valores associados às variáveis x e y, usando uma célula de memória alocada dinamicamente como variável auxiliar. Mais uma vez, estamos a considerar que $\mathbf{F}ields = \{valor\}$ e que $\mathbf{type}(valor) = Int$.

$\texttt{new}(p)$;
$p \to valor := x$;
$x := y$;
$y := p \to valor$;
$\texttt{dispose}(p)$

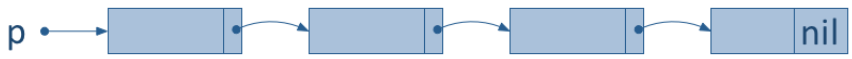
Vejamos agora as alterações que este programa provoca quer na *store* quer na *heap*. Consideremos que partimos de um estado em que a variável x tem o valor 1 e y tem o valor 2, e a variável de endereço p tem um valor que não interessa, 5 digamos, e em que a *heap* é vazia.

Comandos:	Estado:
	$\mathbf{S}tore: \quad x:1,\ y:2,\ p:5$
	$\mathbf{H}eap:$
$\texttt{new}(p)$	
	$\mathbf{S}tore: \quad x:1,\ y:2,\ p:3$
	$\mathbf{H}eap: \quad 3 \mapsto [valor: -]$
$p \to valor := x$	
	$\mathbf{S}tore: \quad x:1,\ y:2,\ p:3$
	$\mathbf{H}eap: \quad 3 \mapsto [valor:1]$
$x := y$	
	$\mathbf{S}tore: \quad x:2,\ y:2,\ p:3$
	$\mathbf{H}eap: \quad 3 \mapsto [valor:1]$
$y := p \to valor$	
	$\mathbf{S}tore: \quad x:2\ ,y:1,\ p:3$
	$\mathbf{H}eap: \quad 3 \mapsto [valor:1]$
$\texttt{dispose}(p)$	
	$\mathbf{S}tore: \quad x:2,\ y:1,\ p:3$
	$\mathbf{H}eap:$

Com a execução do comando $\texttt{new}(p)$ é alocado na *heap* um *record* , sendo o endereço guardado em p. Como não é feita inicialização do *record* , o conteúdo do campo *valor* pode ser qualquer. Vamos representar isso por - .
De seguida é executado o comando $p \to valor := x$ que coloca no campo *valor* do endereço p o valor que a variável x tem, ou seja, o valor 1.

Posteriormente é feita uma atribuição e a variável x passa a conter o valor da variável y, ou seja, o valor 2. Com recurso ao comando $y := p \rightarrow valor$, é atualizado o valor da variável y com o valor que se encontrava no endereço p no campo valor. Por fim, como já foi feita a troca de valores, podemos libertar a memória que foi alocada para permitir esta troca. Assim, com recurso ao comando `dispose`(p) é libertado o endereço de p na *heap*.

Exemplo 2.2.7 O conceito de lista ligada é muito importante em programação. De seguida, pretendemos ilustrar, com um exemplo, a utilização de listas ligadas, na linguagem $While_{pr}$. Assumimos que $\mathbf{F}ields = \{valor, seg\}$ onde $\mathbf{type}(valor) = Int$ e $\mathbf{type}(seg) = Loc$. O código abaixo permite contar os elementos de uma lista ligada, devolvendo esse valor na variável n. A variável n terá o número de nodos da lista l que já foram contados e p é um apontador auxiliar que vai percorrendo a lista de forma que se possam contar os seus nodos. Portanto, n é inicializado a zero e p com o valor de l. Enquanto não se atingir o fim da lista (codificado na condição $p \neq nil$) o valor de n é incrementado e p passa a apontar para a célula seguinte da lista. Daí que, quando o ciclo *while* terminar, n tem o número total de células da lista ligada.



$p := l;$
$n := 0;$
`while`$\neg(p = nil)\{$
  $n := n + 1;$
  $p := p \rightarrow seg;$
$\}$

Apesar de muito simples a linguagem $While_{pr}$ permite-nos lidar com estruturas dinâmicas de forma muito similar a outras linguagens imperativas com apontadores conhecidas.

Capítulo 3

Lógica da Separação para a linguagem $While_{pr}$

A ideia de especificar o comportamento de programas através do uso de asserções que são avaliadas (como verdadeiras ou falsas), em relação ao estado de execução de um programa, foi introduzida por C.A.R Hoare [14, 13], em 1969, para a linguagem *While*.

A lógica de Hoare permite raciocinar acerca da correcção de programas imperativos, baseando-se na lógica de 1ª ordem para descrever o estado de execução dos programas. Neste contexto, a ideia de triplo de Hoare $\{\phi\}\mathsf{C}\{\psi\}$ assume um papel fulcral, e o seu significado intuitivo é o de que se o programa *C* for executado a partir de um estado inicial onde a asserção ϕ (a pré-condição) é válida, então, se o programa terminar, a asserção ψ (a pós-condição) é válida no seu estado final [3].

A utilização da lógica de Hoare para raciocinar com programas que manipulam memória dinâmica, embora possível, revela-se muito pouco adequada, dando origem a asserções complicadas, com as quais é muito penoso lidar [21].

No final do século *XX*, P. O'Hearn e J. Reynolds [21, 19, 23] apresentaram uma extensão da lógica de Hoare que permite raciocinar de maneira mais simples com programas que manipulam memória dinâmica. O aspecto essencial desta abordagem é a utilização dos triplos de Hoare com uma extensão da linguagem de asserções em que se destaca uma nova conectiva - a *conjunção separada* - cuja semântica é a de que as suas componentes são válidas em partes disjuntas da heap. A esta extensão da lógica de Hoare deu-se o nome de *lógica da separação*. Esta será a lógica usada para verificar programas da linguagem $While_{pr}$.

A par da semântica, introduziremos um sistema de inferência **S**, para definir os triplos dedutíveis da lógica da separação e provaremos a correcção do sistema **S** em relação à semântica.

As opções relativas à linguagem $While_{pr}$, apresentadas no capítulo anterior, irão reflectir-se neste capítulo e, terão por consequência, em particular, um sistema de inferência diferente do apresentado em [21, 19].

3.1 Linguagem de asserções

A linguagem de asserções (também chamada de anotações) é uma extensão da linguagem da lógica de 1ª ordem.

$$\begin{aligned}&\mathbf{A}ssert \ni \phi, \psi, \theta ::= b \\ &\mid \neg\phi \mid \phi \wedge \psi \mid \phi \vee \psi \mid \phi \Rightarrow \psi \mid \phi \Leftrightarrow \psi \mid \forall v.\phi \mid \exists v.\phi \\ &\mid emp \mid p \mapsto r \mid \phi * \psi \mid \phi \mathrel{-\!\!*} \psi\end{aligned}$$

Figura 3.1: Classe das Asserções

A Figura 3.1 descreve a linguagem de asserções que serão usadas para especificar as pré e pós-condições para os programas. As asserções vão permitir-nos caracterizar o estado, que, como já vimos, tem duas componentes: a *store* e a *heap*. Algumas vezes, em vez da palavra asserção usaremos a palavra fórmula.
Para além das usuais fórmulas em lógica de 1ª ordem (recorde-se que b representa uma expressão booleana, conforme Figura 2.1), temos quatro novas formas de asserção para descrever a heap:

- emp - indica que a heap é vazia.
- $p \mapsto r$ - denota que a *heap* tem uma única célula alocada e que essa célula está no endereço p e o seu conteúdo é o *record* r.
- $\phi * \psi$ - indica que a *heap* pode ser partida em duas partes disjuntas, sendo ϕ válida numa parte da *heap* e ψ válida na outra parte.
- $\phi \mathrel{-\!\!*} \psi$ - indica que se a *heap* actual pode ser estendida com uma parte disjunta que valida ϕ, então ψ será válida na *heap* estendida.

$$\mathbf{R}ecords \ni r ::= [f_1 : a_1, ..., f_n : a_n] \text{ com } \mathbf{F}ields = \{f_1, ..., f_n\} \text{ e } a_i \in \mathbf{Exp}_{\mathbf{type}(f_i)} \text{ para } i \in \{1, ..., n\}$$

Figura 3.2: Classe de records

Por razões de simplificação da exposição, o nosso conjunto de asserções atómicas é fixo, permitindo apenas b, emp e $p \mapsto r$. No entanto, para a apresentação de alguns exemplos, surgiu a necessidade de considerar um conjunto mais rico de fórmulas atómicas e consequentemente assumir extensões adequadas das definições que se vão seguir.

Notação 3.1.1 Adoptamos as seguintes abreviaturas para asserções:

- $p \mapsto [f : a] = \exists v_1 ... \exists v_n . p \mapsto [f_1 : v_1, ..., f : a, ... f_n : v_n]$

- $p \mapsto [f : -] = \exists v.p \mapsto [f : v]$
- $p \mapsto [\,-\,] = \exists v_1 ... \exists v_n.p \mapsto [f_1 : v_1, ..., f_n : v_n]$
- $p \hookrightarrow r = p \mapsto r * \top$

Notação 3.1.2 Alguma notação sobre *heaps*

- $h = [\ ]$ significa que a *heap* h se encontra vazia
- $h_1 \# h_2$ indica que h_1 e h_2 são *heaps* disjuntas, ou seja, $dom(h_1) \cap dom(h_2) = \{\}$
- $h_1 \,.\, h_2$ denota a união de duas *heaps* h_1 e h_2, tal que $h_1 \# h_2$

No Capítulo 1 já apresentámos a interpretação das várias expressões da linguagem $While_{pr}$. Recorde-se que esta interpretação depende apenas da *store* (não depende da *heap*). É agora necessário fixar a interpretação das várias asserções, em particular, das novas asserções relacionadas com a manipulação da heap. A interpretação das asserções será feita no contexto de um estado (*store* e *heap*).

Definição 3.1.3 A interpretação de uma fórmula é dada pela função $[\![\phi]\!] : \mathbf{S}tate \rightarrow \{T, F\}$ definida recursivamente da seguinte forma:
$[\![b]\!](s,h) = [\![b]\!](s)$
$[\![\neg\phi]\!](s,h) = T \Leftrightarrow [\![\phi]\!](s,h) = F$
$[\![\phi \wedge \psi]\!](s,h) = T \Leftrightarrow [\![\phi]\!](s,h) = T \wedge [\![\psi]\!](s,h) = T$
$[\![\phi \vee \psi]\!](s,h) = T \Leftrightarrow [\![\phi]\!](s,h) = T \vee [\![\psi]\!](s,h) = T$
$[\![\phi \Rightarrow \psi]\!](s,h) = F \Leftrightarrow [\![\phi]\!](s,h) = T \wedge [\![\psi]\!](s,h) = F$
$[\![\phi \Leftrightarrow \psi]\!](s,h) = T \Leftrightarrow [\![\phi]\!](s,h) = [\![\psi]\!](s,h)$
$[\![\exists v.\phi]\!](s,h) = T \Leftrightarrow [\![\phi]\!]([s|v : a],h) = T$ para algum $a \in \mathbf{V}alues$
$[\![\forall v.\phi]\!](s,h) = T \Leftrightarrow [\![\phi]\!]([s|v : a],h) = T$ para qualquer $a \in \mathbf{V}alues$
$[\![emp]\!](s,h) = T \Leftrightarrow h = [\ ]$
$[\![\phi * \psi]\!](s,h) = T \Leftrightarrow \exists\, h_0, h_1 : h_0 \# h_1 \wedge h_0.h_1 = h \wedge [\![\phi]\!](s,h_0) = T \wedge [\![\psi]\!](s,h_1) = T$
$[\![\phi \mathrel{-\!\!*} \psi]\!](s,h) = T \Leftrightarrow \forall\, h_1 : h_1 \# h \wedge [\![\phi]\!](s,h_1) = T \Rightarrow [\![\psi]\!](s,h_1.h) = T$
$[\![p \mapsto [f_i : a_i]]\!](s,h) = T \Leftrightarrow dom(h) = \{[\![p]\!](s)\} \wedge h([\![p]\!](s))(f_i) = [\![a_i]\!](s)$

Proposição 3.1.4 Em consequência da introdução da definição anterior, a interpretação das abreviaturas apresentadas em 3.1.1 é respectivamente:
$[\![p \mapsto [-]]\!](s,h) = T \Leftrightarrow dom(h) = \{[\![p]\!](s)\}$
$[\![p \mapsto [f : a]]\!](s,h) = T \Leftrightarrow dom(h) = \{[\![p]\!](s)\} \wedge h([\![p]\!](s))(f) = [\![a]\!](s)$
$[\![p \mapsto [f : -]]\!](s,h) = T \Leftrightarrow dom(h) = \{[\![p]\!](s)\} \wedge \exists a \in \mathbf{V}alues : h([\![p]\!](s))(f) = a$
$[\![p \hookrightarrow [f_i : a_i]]\!](s,h) = T \Leftrightarrow [\![p]\!](s) \in dom(h) \wedge (h([\![p]\!](s)))(f_i) = [\![a_i]\!](s)$

Daqui em diante, na interpretação de uma asserção ϕ, para tornar a notação mais leve, em vez de escrevermos $[\![\phi]\!](s,h) = T$ (respectivamente, $[\![\phi]\!](s,h) = F$), escreveremos simplesmente $[\![\phi]\!](s,h)$ (respectivamente, $\neg[\![\phi]\!](s,h)$).

Definição 3.1.5 Dizemos que ϕ é válida, escrevendo $\models \phi$, quando ϕ é válida em qualquer estado, ou seja, $\forall (s,h) \in \mathbf{S}tate, [\![\phi]\!](s,h)$.

Por vezes, escreveremos simplesmente uma fórmula para significar que a fórmula é válida quando falarmos acerca de uma propriedade de equivalência ou equivalência lógica (e.g, $\phi \Leftrightarrow \psi$ abreviará $\models \phi \Leftrightarrow \psi$).

Consideremos que $Vars(e)$ denota o conjunto de variáveis de e quando e é uma expressão (inteira, endereço ou *booleana*) ou um *record*.

Definição 3.1.6 $free(\phi)$ denota o conjunto das variáveis livres de ϕ (uma noção adaptada de [16]) e é definido recursivamente por:
$free(b) = Vars(b)$
$free(\neg\phi) = free(\phi)$
$free(\phi \square \psi) = free(\phi) \cup free(\psi)$ com $\square \in \{\wedge, \vee, \Rightarrow, \Leftrightarrow, *, -\!\!*\}$
$free(Qv.\phi) = free(\phi) \setminus \{v\}$ com $Q \in \{\forall, \exists\}$
$free(emp) = \{\}$
$free(p \mapsto r) = Vars(p) \cup Vars(r)$

Para as provas apresentadas ao longo da dissertação serão necessárias algumas propriedades sobre as novas asserções que indicamos de seguida. Estas propriedades encontram-se enunciadas também em [21, 23] :

Proposição 3.1.7 As seguintes asserções são válidas:

1. $\phi * \psi \Leftrightarrow \psi * \phi$
2. $(\phi * \theta) * \psi \Leftrightarrow \phi * (\theta * \psi)$
3. $\phi * emp \Leftrightarrow \phi$
4. $(\phi \vee \psi) * \theta \Leftrightarrow (\phi * \theta) \vee (\psi * \theta)$
5. $(\phi \vee \psi) * \theta \Rightarrow (\phi * \theta) \vee (\psi * \theta)$
6. $(\exists v.\phi) * \psi \Leftrightarrow \exists v.(\phi * \psi)$ quando $v \notin free(\psi)$
7. $(\forall v.\phi) * \psi \Leftrightarrow \forall v.(\phi * \psi)$ quando $v \notin free(\psi)$

Demonstração A título ilustrativo provemos a 1ª propriedade. Suponhamos $[\![\phi * \psi]\!](s,h)$, ou seja, $\exists h_1, h_2 : h_1 \# h_2 \wedge h = h_1.h_2 \wedge [\![\phi]\!](s,h_1) \wedge [\![\psi]\!](s,h_2)$.

Pretendemos demonstrar $[\![\psi * \phi]\!](s,h)$, ou seja, $\exists h_3, h_4 : h_3 \# h_4 \wedge h = h_3.h_4 \wedge [\![\phi]\!](s,h_3) \wedge [\![\psi]\!](s,h_4)$. Para tal, basta tomar, $h_3 = h_2$ e $h_4 = h_1$.

□

Nas provas, usaremos muitas vezes o seguinte lema de substituição (sem fazer referência explícita ao Lema):

Lema 3.1.8 $\forall \phi \in \mathbf{A}ssert, (s,h) \in \mathbf{S}tate\ [\![\phi]\!]([s|v : [\![e]\!](s)], h) = [\![\phi[e/v]]\!](s,h)$

3.2 Triplos de Hoare

Estamos agora em condições de apresentar o conceito central desta lógica de programas e que é o conceito de *especificação* ou *triplo de Hoare*.

$$\mathbf{T}riplo \ni T ::= \{\phi\}\mathsf{C}\{\psi\}$$

Figura 3.3: Classe das especificação ou Triplos de Hoare

Na Figura 3.3 descreve-se a sintaxe dos triplos de Hoare. O significado intuitivo de um triplo $\{\phi\}\mathsf{C}\{\psi\}$ é o de que se o programa C for executado num estado inicial em que a asserção ϕ (a pré-condição) seja válida, então, ou a execução de C não termina, ou, se terminar, não produz um erro (*abort*) e a asserção ψ (a pós-condição) será válida no estado final. A esta semântica de triplos dá-se o nome de especificação de correcção parcial (dado que a propriedade de terminação do programa não é garantida).

Definição 3.2.1 Dizemos que um triplo $\{\phi\}\mathsf{C}\{\psi\}$ é válido, e escrevemos $\models \{\phi\}\mathsf{C}\{\psi\}$, quando $\forall(s,h) \in \mathbf{S}tate$ se $[\![\phi]\!](s,h)$ então:

1. $safe(C,(s,h))$
2. $\forall(s',h') \in \mathbf{S}tate$ se $(C,(s,h)) \rightsquigarrow^* (\mathtt{skip},(s',h'))$ então $[\![\psi]\!](s',h')$

A validade de um triplo $\{\phi\}\mathsf{C}\{\psi\}$ implica portanto que a execução do programa C a partir de um estado que valida a pré-condição ϕ, nunca pode abortar (condição 1). Se a execução do programa der um erro de acesso à *heap* (e portanto *abort*), então o triplo será inválido.

3.3 O sistema dedutivo S

Nesta secção introduziremos um sistema dedutivo para triplos de Hoare, que designaremos por Sistema **S**, que pode ser visto como um conjunto de leis fundamentais acerca dos programas. O sistema dedutivo para a lógica da separação está apresentado na Figura 3.4. Este sistema contém uma regra para cada comando da linguagem e três regras estruturais (consequência, *frame* e eliminação de variáveis auxiliares). Analisando as regras apresentadas, podemos dizer que as regras referentes aos comandos *skip*, atribuição, sequenciação, condicional e ciclo *while* são exactamente iguais às da lógica de Hoare. Os comandos de manipulação da *heap* dão origem a quatro axiomas:

- *Mutation* - especifica a alteração de um campo (f) da única célula (p) da heap
- *Dispose* - especifica a libertação da única célula (p) da heap

- *Allocation* - especifica a alocação de uma célula (p) numa heap vazia
- *Lookup* - especifica a atribuição de um valor de memória (que se encontra na célula p no campo f) para uma variável (v)

Das regras estruturais, para além das regras da consequência e de eliminação da variáveis auxiliares já conhecidas da lógica de Hoare, surge agora uma nova regra a que se dá o nome de *Frame*. De forma intuitiva, a regra de *Frame* permite introduzir uma parte disjunta da heap nas pré e pós-condição, onde a execução do comando C não provoca nenhuma alteração. De forma mais rigorosa, a regra de *Frame* usa uma condição lateral, que garante a não interferência das variáveis alteradas pela execução do comando C e das variáveis livres de θ como forma de assegurar a validade da asserção θ. Tal é conseguido com recurso à conjunção separada.

O sistema formal **S** será a base da metodologia seguida para a verificação de programas $While_{pr}$ descrita no capítulo seguinte. Provaremos agora a correcção deste sistema em relação à semântica de triplos de Hoare introduzida na secção anterior.

Definição 3.3.1 $modifies(\mathsf{C})$ denota o conjunto de variáveis (de tipo inteiro ou de endereço) que são modificadas pelo comando C. $free(\mathsf{C})$ denota o conjunto das variáveis livres de C.(As definições abaixo seguem [16])

1. $modifies(\mathsf{C})$ é definido recursivamente por:
   $modifies(\texttt{skip}) = \{\}$
   $modifies(v{:=}a) = \{v\}$
   $modifies(\mathsf{C}_1;\mathsf{C}_2) = modifies(\mathsf{C}_1) \cup modifies(\mathsf{C}_2)$
   $modifies(\texttt{if}\ b\ \texttt{then}\ \mathsf{C}_t\ \texttt{else}\ \mathsf{C}_f) = modifies(\mathsf{C}_t) \cup modifies(\mathsf{C}_f)$
   $modifies(\texttt{while}\ b\ \{\mathsf{C}\}) = modifies(\mathsf{C})$
   $modifies(p{\rightarrow}f{:=}a) = \{\}$
   $modifies(\texttt{dispose}(p)) = \{\}$
   $modifies(\texttt{new}(p)) = \{p\}$
   $modifies(v{:=}p \rightarrow f) = \{v\}$

2. $free(\mathsf{C})$ é definido recursivamente por:
   $free(\texttt{skip}) = \{\}$
   $free(v{:=}a) = \{v\} \cup Vars(a)$
   $free(\mathsf{C}_1;\mathsf{C}_2) = free(\mathsf{C}_1) \cup free(\mathsf{C}_2)$
   $free(\texttt{if}\ b\ \texttt{then}\ \mathsf{C}_t\ \texttt{else}\ \mathsf{C}_f) = free(b) \cup free(\mathsf{C}_t) \cup free(\mathsf{C}_f)$
   $free(\texttt{while}\ b\ \{\mathsf{C}\}) = free(b) \cup free(\mathsf{C})$
   $free(p{\rightarrow}f{:=}a) = \{p\} \cup Vars(a)$
   $free(\texttt{new}(p)) = \{p\}$
   $free(\texttt{dispose}(p)) = \{p\}$
   $free(v{:=}p \rightarrow f) = \{v, p\}$

Como é usual, uma dedução (também chamada derivação) no sistema **S** será uma árvore de triplos construída a partir dos axiomas (regras sem premissas), por aplicação das outras regras de inferência.

$$\frac{}{\{\phi\}\texttt{skip}\{\phi\}}\ \text{Skip}$$

$$\frac{}{\{\psi[a/v]\}v{:=}a\{\psi\}}\ \text{Atrib.}$$

$$\frac{\{\phi\}\mathsf{C}_1\{\theta\} \qquad \{\theta\}\mathsf{C}_2\{\psi\}}{\{\phi\}\mathsf{C}_1;\mathsf{C}_2\{\psi\}}\ \text{Comp.}$$

$$\frac{\{\phi \wedge b\}\mathsf{C}_t\{\psi\} \qquad \{\phi \wedge \neg b\}\mathsf{C}_f\{\psi\}}{\{\phi\}\texttt{if}\ b\ \texttt{then}\ \mathsf{C}_t\ \texttt{else}\ \mathsf{C}_f\{\psi\}}\ \text{If}$$

$$\frac{\{\theta \wedge b\}\mathsf{C}\{\theta\}}{\{\theta\}\texttt{while}\ b\ \{\mathsf{C}\}\{\theta \wedge \neg b\}}\ \text{While}$$

$$\frac{}{\{p \mapsto [f:-]\}p{\to}f{:=}a\{p \mapsto [f:a]\}}\ \text{Mutation}$$

$$\frac{}{\{p \mapsto [-]\}\texttt{dispose}(p)\{emp\}}\ \text{Dispose}$$

$$\frac{}{\{emp\}\texttt{new}(p)\{p \mapsto [-]\}}\ \text{Allocation}$$

$$\frac{v \notin Vars(a) \cup Vars(m)}{\{v = m \wedge p \mapsto [f:a]\}v{:=}p \to f\{v = a \wedge p[m/v] \mapsto [f:a]\}}\ \text{Lookup}$$

$$\frac{\{\phi\}\mathsf{C}\{\psi\} \qquad \models \phi' \Rightarrow \phi \qquad \models \psi \Rightarrow \psi'}{\{\phi'\}\mathsf{C}\{\psi'\}}\ \text{Conseq.}$$

$$\frac{\{\phi\}\mathsf{C}\{\psi\} \qquad modifies(\mathsf{C}) \cap free(\theta) = \varnothing}{\{\phi * \theta\}\mathsf{C}\{\psi * \theta\}}\ \text{Frame}$$

$$\frac{\{\phi\}\mathsf{C}\{\psi\} \qquad v \notin free(\mathsf{C})}{\{\exists v.\phi\}\mathsf{C}\{\exists v.\psi\}}\ \text{Aux. Var. Elim.}$$

Figura 3.4: Regras de inferência do Sistema **S**

Definição 3.3.2 Dizemos que $\{\phi\}\mathsf{C}\{\psi\}$ é dedutível (ou derivável) no Sistema **S**, e escrevemos $\vdash_S \{\phi\}\mathsf{C}\{\psi\}$, quando existe alguma dedução (derivação) cuja conclusão (raiz da árvore) é o triplo $\{\phi\}\mathsf{C}\{\psi\}$.

Exemplo 3.3.3 Consideremos de novo o Exemplo 4.3.3, agora enriquecido com uma pré e pós condição, que reflectem a troca entre os valores das variáveis x e y.

$\{x = 1 \ \wedge \ y = 2\}$
$\texttt{new}(p);$
$p \rightarrow \ valor := x;$
$x := y;$
$y := p \ \rightarrow \ valor;$
$\texttt{dispose}(p);$
$\{x = 2 \ \wedge \ y = 1\}$

Este triplo é dedutível no Sistema **S**, vejamos como:

$$D_1 = \dfrac{\dfrac{\dfrac{}{\{emp\}\texttt{new}(p)\{p \mapsto [-]\}}\,\text{Allocation}}{\{(x = 1 \wedge y = 2) * emp\}\texttt{new}(p)\{(x = 1 \wedge y = 2) * p \mapsto [-]\}}\,\text{Frame}}{\{x = 1 \wedge y = 2\}\texttt{new}(p)\{(x = 1 \wedge y = 2) * p \mapsto [-]\}}\,\text{Conseq.}$$

$$D_2 = \dfrac{\dfrac{\dfrac{}{\{p \mapsto [-]\}p \rightarrow valor := x\{p \mapsto [valor : x]\}}\,\text{Mutation}}{\{(x = 1 \wedge y = 2) * p \mapsto [-]\}p \rightarrow valor := x\{(x = 1 \wedge y = 2) * p \mapsto [valor : x]\}}\,\text{Frame}}{\{(x = 1 \wedge y = 2) * p \mapsto [-]\}p \rightarrow valor := x\{y = 2 * p \mapsto [valor : 1]\}}\,\text{Conseq.}$$

$$D_3 = \dfrac{\dfrac{}{\{\exists m.(y = m \wedge y = 2) * p \mapsto [valor : 1]\}x := y\{\exists m.(x = 2 \wedge y = m) * p \rightarrow [valor : 1]\}}\,\text{Atrib.}}{\{y = 2 * p \mapsto [valor : 1]\}x := y\{\exists m.(x = 2 \wedge y = m) * p \rightarrow [valor : 1]\}}\,\text{Conseq.}$$

$$D_4 = \dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\{\exists m.y = m \wedge p \rightarrow [valor : 1]\}y := p \rightarrow valor\{y = 1 \wedge p \mapsto [valor : 1]\}}\,\text{Lookup}}{\{\exists m.y = m \wedge p \rightarrow [valor : 1]\}y := p \rightarrow valor\{\exists a.y = 1 \wedge p \mapsto [valor : a]\}}\,\text{Conseq.}}{\{\exists m.x = 2 * (y = m \wedge p \rightarrow [valor : 1])\}y := p \rightarrow valor\{x = 2 * \exists a.(y = 1 \wedge p \mapsto [valor : a])\}}\,\text{Frame}}{\{\exists m.(x = 2 \wedge y = m) * p \rightarrow [valor : 1]\}y := p \rightarrow valor\{\exists a.(x = 2 \wedge y = 1) * p \mapsto [valor : a]\}}\,\text{Conseq.}}{\{\exists m.(x = 2 \wedge y = m) * p \rightarrow [valor : 1]\}y := p \rightarrow valor\{(x = 2 \wedge y = 1) * p \mapsto [\,]\}}\,\text{Conseq.}$$

$$D_5 = \dfrac{\dfrac{\dfrac{}{\{p \mapsto [-]\}\texttt{dispose}(p)\{emp\}}\,\text{Dispose}}{\{(x = 2 \wedge y = 1) * p \mapsto [-]\}\texttt{dispose}(p)\{(x = 2 \wedge y = 1) * emp\}}\,\text{Frame}}{\{(x = 2 \wedge y = 1) * p \mapsto [-]\}\texttt{dispose}(p)\{x = 2 \wedge y = 1\}}\,\text{Conseq.}$$

$$D = \dfrac{D_1 \qquad \dfrac{D_2 \qquad \dfrac{D_3 \qquad \dfrac{D_4 \qquad D_5}{\text{Comp.}}}{\text{Comp.}}}{\text{Comp.}}}{\{x = 1 \wedge y = 2\}\texttt{new}(p); p \rightarrow valor := x; x := y; y := p \rightarrow valor; \texttt{dispose}(p)\{x = 2 \wedge y = 1\}}\,\text{Comp.}$$

As condições laterais relativas às aplicações da regra da consequência, provam-se facilmente com recurso à Propriedade 3.1.7. Daqui em diante, quando pretendermos construir derivações como a anterior, representa-las-emos da seguinte forma:

$\{x = 1 \wedge y = 2\}$
`new`(p); — Conseq.+ Frame + Allocation
$\{(x = 1 \wedge y = 2) * p \mapsto [-]\}$
$p \rightarrow valor := x$; — Conseq. + Frame + Mutation
$\{y = 2 * p \mapsto [valor : 1]\}$
$x := y$; — Conseq. + Atrib.
$\{\exists m.(x = 2 \wedge y = m) * p \rightarrow [valor : 1]\}$
$y := p \rightarrow valor$; — Conseq. + Conseq. + Frame + Conseq. +Lookup
$\{(x = 2 \wedge y = 1) * p \mapsto [\,]\}$
`dispose`(p) — Conseq. + Frame + Dispose
$\{x = 2 \wedge y = 1\}$

Não será mencionada explicitamente a aplicação da regra de composição, mas quando escrevermos uma estrutura desta forma $\{\phi\}C\{\phi'\}C'\{\phi''\}$ estaremos implicitamente a aplicar a regra da composição. Surgem por vezes estruturas do tipo $\{\phi\}\{\phi'\}C\{\phi'\}$ que significam a aplicação de uma regra estrutural, sendo esta mencionada à frente da segunda asserção.

Terminamos esta secção com a prova da correcção do sistema **S** relativamente à semântica dos triplos de Hoare. É importante mencionar que, devido à dimensão desta prova, esta foi divida e em anexo são apresentados alguns casos.

Teorema 3.3.4 Se $\vdash_S \{\phi\}C\{\psi\}$, então $\models \{\phi\}C\{\psi\}$.

Demonstração Aplicando o princípio de indução associado às deduções de **S**.

1. Caso *mutation*:

   É necessário garantir que $\{p \mapsto [f : -]\}p{\rightarrow}f{:=}a\{p \mapsto [f : a]\}$ é válido, ou seja, supondo $[\![p \mapsto [f : -]]\!](s,h)$ (que implica $dom(h) = \{[\![p]\!](s)\}$), temos que mostrar:

   (a) $safe(p{\rightarrow}f{:=}a,(s,h))$

   (b) $\forall (s',h') \in \mathbf{S}tate$ se $(p{\rightarrow}f{:=}a,(s,h)) \rightsquigarrow^* (\mathtt{skip},(s',h'))$ então $[\![p \mapsto [f : a]]\!](s',h')$

   (a) $safe(p{\rightarrow}f{:=}a,(s,h))$, pois $(p{\rightarrow}f{:=}a,(s,h)) \rightsquigarrow^* abort$ apenas se $[\![p]\!](s) \notin dom(h)$ e esta última condição é falsa.

   (b) Atendendo à definição de $\rightsquigarrow^*$, às transições possíveis por $\rightsquigarrow$ com o comando *mutation* e ao facto de não existirem transições partir do comando *skip*,

   $$(p{\rightarrow}f{:=}a,(s,h)) \rightsquigarrow^* (\mathtt{skip},(s',h'))$$

   implica $s' = s$, $h' = [h|[\![p]\!](s) : r]$ e $r = [h([\![p]\!](s))|f : [\![a]\!](s)]$. Queremos demonstrar que $[\![p \mapsto [f : a]]\!](s',h')$. Ora,

$$
\begin{array}{l}
[\![p \mapsto [f:a]]\!](s',h') \\
\Leftrightarrow [\![p \mapsto [f:a]]\!](s,[h|[\![p]\!](s):r]) \\
\Leftrightarrow dom([h|[\![p]\!](s):r]) = \{[\![p]\!](s)\} \wedge [h|[\![p]\!](s):r]([\![p]\!](s))(f) = [\![a]\!](s)
\end{array}
$$

De sabermos que $dom(h) = \{[\![p]\!](s)\}$, temos que $dom([h|[\![p]\!](s):r]) = dom(h) \cup \{[\![p]\!](s)\} = \{[\![p]\!](s)\} \cup \{[\![p]\!](s)\} = \{[\![p]\!](s)\}$.
Falta agora demonstrar que $[h|[\![p]\!](s):r]([\![p]\!](s))(f) = [\![a]\!](s)$. Ora, $[h|[\![p]\!](s):r]([\![p]\!](s))(f) = r(f) = [\![a]\!](s)$.

2. Caso *allocation*:
Pretendemos demonstrar a validade do triplo $\{emp\}\texttt{new}(p)\{p \mapsto [-]\}$, ou seja, supondo $[\![emp]\!](s,h)(\Leftrightarrow h = [\ ])$ temos que garantir,

   (a) $safe(\texttt{new}(p),(s,h))$

   (b) $\forall (s',h') \in \mathbf{S}tate$ se $(\texttt{new}(p),(s,h)) \rightsquigarrow^* (\texttt{skip},(s',h'))$ então $[\![p \mapsto [-]]\!](s',h')$

   (a) Temos que mostrar que $(\texttt{new}(p),(s,h))\neg \rightsquigarrow^* abort$. Atendendo à definição de $\rightsquigarrow$, uma computação para *abort* nunca é possível com o comando $\texttt{new}(p)$.

   (b) Da definição de $\rightsquigarrow^*$ e de $\rightsquigarrow$, sabemos que

   $$(\texttt{new}(p),(s,h)) \rightsquigarrow^* (\texttt{skip},(s',h'))$$

   implica $s' = [s|p:l]$ e $h' = [h|l:r]$ para algum $l \in \mathbf{A}ddress$ e para algum $r \in record$. Queremos demonstrar que $[\![p \mapsto [-]]\!](s',h')$. Ora,

   $$
   \begin{array}{l}
   [\![p \mapsto [-]]\!](s',h') \\
   \Leftrightarrow [\![p \mapsto [-]]\!]([s|p:l],[h|l:r]) \\
   \Leftrightarrow dom([h|l:r])) = \{l\}
   \end{array}
   $$

   De $h = [\,]$ sabemos que $dom(h) = \{\}$ e portanto, $dom([h|l:r])) = dom(h) \cup \{l\} = \{\} \cup \{l\} = \{l\}$.

3. Caso *dispose*:
É necessário demonstrar a validade de $\{p \mapsto [-]\}\texttt{dispose}(p)\{emp\}$, ou seja, supondo que $[\![p \mapsto [-]]\!](s,h)(\Leftrightarrow dom(h) = \{[\![p]\!](s)\})$ queremos demonstrar que

   (a) $safe(\texttt{dispose}(p),(s,h))$

   (b) $\forall (s',h') \in \mathbf{S}tate$ se $(\texttt{dispose}(p),(s,h)) \rightsquigarrow^* (\texttt{skip},(s',h'))$ então $[\![emp]\!](s',h')$.

(a) Temos que mostrar que $(\texttt{dispose}(p),(s,h)) \not\rightsquigarrow^* abort$. Atendendo à definição de $\rightsquigarrow$, existe uma computação para *abort* se $[\![p]\!](s) \notin dom(h)$, mas isto contradiz a suposição inicial que permite concluir $dom(h) = \{[\![p]\!](s)\}$.

(b) Atendendo às definições de $\rightsquigarrow$ e $\rightsquigarrow^*$, e sabendo que $[\![p]\!](s) \in dom(h)$,

$$(\texttt{dispose}(p),(s,h)) \rightsquigarrow (\texttt{skip},(s',h'))$$

implica $s' = s$ e $h' = h\rceil(dom(h) - \{[\![p]\!](s)\})$. Queremos demonstrar que $[\![emp]\!](s',h')$. Ora,

$$\begin{array}{l}[\![emp]\!](s',h') \\ \Leftrightarrow [\![emp]\!](s,h\rceil(dom(h) - \{[\![p]\!](s)\})) \\ \Leftrightarrow h\rceil(dom(h) - \{[\![p]\!](s)\}) = [\,]\end{array}$$

Mas, de sabermos que $dom(h) = \{[\![p]\!](s)\}$, temos que $h\rceil(\{[\![p]\!](s)\} - \{[\![p]\!](s)\}) = h\rceil\{\} = [\,]$.

4. Caso *lookup*:
É necessário demonstrar a validade do triplo

$$\{v = m \wedge p \mapsto [f : a]\}v{:=}p \rightarrow f\{v = a \wedge p[m/v] \mapsto [f : a]\},$$

ou seja, supondo que $[\![v = m \wedge p \mapsto [f : a]]\!](s,h) (\Leftrightarrow [\![v = m]\!](s,h) \wedge dom(h) = \{[\![p]\!](s)\} \wedge h([\![p]\!](s))(f) = [\![a]\!](s)$, queremos demonstrar que

(a) $safe(v{:=}p \rightarrow f,(s,h))$

(b) $\forall (s',h') \in \mathbf{S}tate$ se $(v{:=}p \rightarrow f,(s,h)) \rightsquigarrow^* (\texttt{skip},(s',h'))$ então $[\![v = a \wedge (p[m/v] \mapsto [f : a])]\!](s',h')$

(a) Queremos mostrar que $(v{:=}p \rightarrow f,(s,h)) \not\rightsquigarrow^* abort$. No entanto, atendendo à definição de $\rightsquigarrow$, a computação para *abort* só é possível se $[\![p]\!](s) \notin dom(h)$, o que contraria $dom(h) = \{[\![p]\!](s)\}$, que segue da suposição inicial.

(b) Supondo que

$$(v{:=}p \rightarrow f,(s,h)) \rightsquigarrow^* (\texttt{skip},(s',h'))$$

temos que $s' = [s|v : (h([\![p]\!](s)))(f)]$ e $h' = h$. Queremos então mostrar $[\![v = a \wedge p[m/v] \mapsto [f : a]]\!](s',h')$. Ora,

$$\begin{array}{l}[\![v = a \wedge p[m/v] \mapsto [f : a]]\!](s',h') \\ \Leftrightarrow [\![v = a]\!](s',h') \wedge [\![p[m/v] \mapsto [f : a]]\!](s',h') \\ \Leftrightarrow [\![v]\!](s') = [\![a]\!](s') \wedge dom(h') = \{[\![p[m/v]]\!](s)\} \wedge (h([\![p[m/v]]\!](s')))(f) = [\![a]\!](s')\end{array}$$

i) $[\![v]\!](s') = [\![a]\!](s') \Leftrightarrow [\![v]\!]([s|v : (h([\![p]\!](s)))(f)]) = [\![a]\!](s') \Leftrightarrow (h([\![p]\!](s)))(f) = [\![a]\!](s')$
(por hipótese) $[\![a]\!](s) = [\![a]\!](s')$ e a última é verdadeira por $v \notin Vars(a)$.

ii)$dom(h') = \{[\![p[m/v]]\!](s)\} \Leftrightarrow$ (por $h' = h$) $dom(h) = \{[\![p]\!]([s'|v : m])\} \Leftrightarrow$
$dom(h) = \{[\![p]\!](s)\}$, que é verdadeiro por suposição.

iii)$(h'([\![p]\!](s')))(f) = [\![a]\!](s') \Leftrightarrow (h([\![p]\!](s')))(f) = [\![a]\!](s') \Leftrightarrow$
$(h([\![p]\!]([s|v : m])))(f) = [\![a]\!](s') \Leftrightarrow [\![a]\!](s') = [\![a]\!](s)$
o que é verdade por $v \notin Vars(a)$.

5. Caso *frame*:
Como hipótese de indução, assumiremos a validade do triplo $\{\phi\}\mathsf{C}\{\psi\}$. Queremos demonstrar a validade do triplo $\{\phi * \theta\}\mathsf{C}\{\psi * \theta\}$, assumindo ainda a condição lateral da regra de *Frame* ($modifies(\mathsf{C}) \cap free(\theta) = \varnothing$).

Supondo $[\![\phi * \theta]\!](s,h)(\Leftrightarrow \exists\, h_0, h_1 \in \mathbf{H}eap :\ h = h_0.h_1 \wedge h_0 \# h_1 \wedge [\![\phi]\!](s,h_0) \wedge [\![\theta]\!](s,h_1))$, queremos demonstrar que:

   (a) $safe(\mathsf{C},(s,h))$

   (b) $\forall (s',h') \in \mathbf{S}tate$ se $(\mathsf{C},(s,h)) \rightsquigarrow^* (\mathtt{skip},(s',h'))$ então $[\![\psi * \theta]\!](s',h')$

   (a) Comecemos por demonstrar que $safe(\mathsf{C},(s,h))$. Ora, de sabermos que $[\![\phi]\!](s,h_0)$, pela hipótese de indução temos que $safe(\mathsf{C},(s,h_0))$. Logo, pelo Lema A.1.17 temos que $safe(\mathsf{C},(s,h_0.h_1))$, isto é, $safe(\mathsf{C},(s,h))$.

   (b) Suponhamos que $(\mathsf{C},(s,h)) \rightsquigarrow^* (\mathtt{skip},(s',h'))$ com $h = h_0.h_1$. Queremos demonstrar que $[\![\psi * \theta]\!](s',h')$. Do Lema A.1.16, como $safe(\mathsf{C},(s,h_0))$, $\exists\, h'' :\ (\mathsf{C},(s,h_0)) \rightsquigarrow^* (\mathtt{skip},(s',h''))$ e $h' = h''.h_1$.

   Como $[\![\phi]\!](s,h_0)$ e $(\mathsf{C},(s,h_0)) \rightsquigarrow^* (\mathtt{skip},(s',h''))$, pela hipótese de indução, $[\![\psi]\!](s',h'')$.

   Como $h' = h''.h_1$, falta apenas provar $[\![\theta]\!](s',h_1)$. Como $modifies(\mathsf{C}) \cap free(\theta) = \varnothing$, temos que s e s' coincidem nos valores atribuídos às variáveis de θ. Logo, $[\![\theta]\!](s',h_1)$ se e só se $[\![\theta]\!](s,h_1)$, e esta última é verdade por suposição.

6. Caso Aux. Var. Elim.:

Como hipótese de indução assumiremos a validade do triplo $\{\phi\}\mathsf{C}\{\psi\}$. Assumiremos que $v \notin free(\mathsf{C})$. Queremos demonstrar a validade do triplo $\{\exists v.\phi\}\mathsf{C}\{\exists v.\psi\}$, ou seja, se $[\![\exists v.\phi]\!](s,h) \Leftrightarrow [\![\phi]\!]([s|v : a],h)$ para algum a, então

   (a) $safe(\mathsf{C},(s,h))$

   (b) $\forall (s',h') \in \mathbf{S}tate$ se $(\mathsf{C},(s,h)) \rightsquigarrow^* (\mathtt{skip},(s',h'))$ então $[\![\exists v.\psi]\!](s',h')$

(a) Comecemos por demonstrar que $safe(\mathsf{C},(s,h))$.
Como $[\![\phi]\!]([s|v:a],h)$ para algum a, aplicando a hipótese de indução temos que $safe(\mathsf{C},([s|v:a],h))$ para algum a. Como $v \notin free(\mathsf{C})$, podemos concluir que $safe(\mathsf{C},(s,h))$.

(b) Falta agora demonstrar que $\forall\,(s',h') \in \mathbf{S}tate$ se $(\mathsf{C},(s,h)) \rightsquigarrow^* (\mathtt{skip},(s',h'))$ então $[\![\exists v.\psi]\!](s',h')$.
Como já vimos, de $[\![\exists v.\phi]\!](s,h)$, temos $[\![\phi]\!]([s|v:a],h)$ para algum a. Por outro lado, assumindo $(\mathsf{C},(s,h)) \rightsquigarrow^* (\mathtt{skip},(s',h'))$, como $v \notin free(\mathsf{C})$, tem-se do Lema A.1.14, também $(\mathsf{C},([s|v:a],h)) \rightsquigarrow (\mathtt{skip},([s'|v:a],h'))$ para algum a. Assim, aplicando a hipótese de indução, temos $[\![\psi]\!]([s'|v:a],h')$ para algum a e portanto $[\![\exists v.\psi]\!](s',h')$.

7. Caso *while*:

Como hipótese de indução temos $\models \{\theta \wedge b\}\mathsf{C}\{\theta\}$.

Queremos demonstrar que se $[\![\theta]\!](s,h)$ então

(a) $safe(\mathtt{while}\ b\ \{\mathsf{C}\},(s,h))$

(b) $\forall (s',h') \in \mathbf{S}tate(\mathtt{while}\ b\ \{\mathsf{C}\},(s,h)) \rightsquigarrow^* (\mathtt{skip},(s',h'))$ então $[\![\theta \wedge \neg b]\!](s',h')$

(a) Comecemos por demonstrar que $safe(\mathtt{while}\ b\ \{\mathsf{C}\},(s,h))$, isto é, $\gamma = (\mathtt{while}\ b\ \{\mathsf{C}\},(s,h)) \not\rightsquigarrow^* abort$. Suponhamos que não. Por indução em n, mostra-se que, para todo (s,h), $safe(\mathsf{C},(s,h))$ e $(\mathtt{while}\ b\ \{\mathsf{C}\},(s,h)) \rightsquigarrow^n abort$ implica uma contradição. O caso $\gamma = abort$ é impossível. Suponhamos que $\gamma \rightsquigarrow \gamma' \rightsquigarrow^* abort$.
Caso $\neg[\![b]\!](s)$, a conclusão é imediata, pois obtemos para γ' a configuração skip.
Caso contrário, $\gamma' = (\mathsf{C};\mathtt{while}\ b\ \{\mathsf{C}\},(s,h))$. Pela hipótese de indução inicial $safe(\mathsf{C},(s,h))$, pelo que para algum s',h' temos $(\mathtt{while}\ b\ \{\mathsf{C}\},(s',h')) \rightsquigarrow^* abort$. Daqui chegamos a uma contradição:

i. se $\neg[\![b]\!](s,)$ obtémos uma configuração de `skip` e

ii. se $[\![b]\!](s)$ teriamos que $safe(\mathsf{C},(s',h'))$ e pela hipótese de indução interna teriamos uma contradição.

(b) Provemos agora que $(\mathtt{while}\ b\ \{\mathsf{C}\},(s,h)) \rightsquigarrow^n (\mathtt{skip},(s',h'))$ então $[\![\theta \wedge \neg b]\!](s',h')$.
A prova segue por indução completa em n: Caso $\neg[\![b]\!](s)$, atendendo à definição de $\rightsquigarrow$, $s' = s$, $h' = h$, pelo que temos $[\![\theta \wedge \neg b]\!](s',h')$. Caso $[\![b]\!](s,h)$, para algum s'',h'' e $n_1,n_2 < n$ teremos:

i. $(\mathsf{C},(s,h)) \rightsquigarrow^{n_1} (\mathtt{skip},(s'',h''))$

ii. $(\mathtt{while}\ b\ \{\mathsf{C}\},(s'',h'')) \rightsquigarrow^{n_2} (\mathtt{skip},(s',h'))$

De $[\![\theta]\!](s,h)$, de $[\![b]\!](s,h)$, de i. e da hipótese de indução inicial $\models \{\theta \wedge b\}\mathsf{C}\{\theta\}$, segue que $[\![\theta]\!](s'',h'')$. Daqui, de ii. e da hipótese de indução interna (note-se que $n_2 < n$), segue então $[\![\theta \wedge \neg b]\!](s',h')$.

□

3.4 Exemplos com listas ligadas

Para poder raciocinar sobre programas que manipulam estruturas de dados dinâmicas, para além da implementação dessas estruturas pelo código do programa, temos que exprimir também essas estruturas em termos abstractos, e de ser capazes de relacionar os estados do programa com os valores abstractos que esses estados denotam.

É, por isso, necessário definir algebricamente o conjunto de valores abstractos com que vamos lidar e definir, indutivamente, os predicados sobre esses valores abstractos que nos permitem exprimir as propriedades que sejam necessárias. Em rigor, a linguagem de anotações tem que ser enriquecida com estes predicados e termos.

Tomemos o exemplo usual das listas ligadas. As listas ligadas são usadas para representar sequências (de comprimento variável). Nas listas ligadas estas sequências são implementadas na *heap* com records com dois campos: $valor$ (onde está guardado o elemento da sequência) e *seg* (que é o endereço do nodo seguinte da sequência). O valor abstracto que está aqui em causa é o de sequência, que pode ser descrito à custa do construtor da sequência vazia ε e do construtor que dado um elemento a e uma sequência α, constrói uma sequência que tem a como primeiro elemento (cabeça) e α como cauda da sequência. A estrutura de dados lista ligada implementa esta ideia representando a sequência vazia pelo apontador nil, e fazendo cada nodo da lista ligada na *heap*, a construção de ligação entre a cabeça e a cauda da sequência.

Vamos usar a seguinte notação sobre sequências:

Notação 3.4.1
- ε denota a sequência vazia.
- $[x]$ denota uma sequência com um único elemento x.
- $\alpha \cdot \beta$ denota a concatenação da lista α com a lista β.
- α^+ denota a lista α invertida
- $\#\alpha$ denota o comprimento da sequência α
- α_i denota a i-ésima componente de α

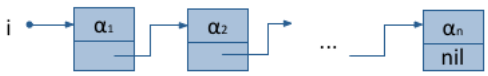


Figura 3.5: Lista ligada: $list(\alpha, i)$

Para descrever a representação de uma lista simplesmente ligada l, como a representa na Figura 3.5, utilizaremos o predicado $list(\alpha, i)$ que pode ser definido indutivamente por:

1. $list(\varepsilon, i) \stackrel{def}{=} emp \wedge i = nil$

2. $list([a] \cdot \alpha, i) \stackrel{def}{=} \exists p. i \mapsto [valor : a, seg : p] * list(\alpha, p)$

Podemos ainda definir um predicado para capturar a noção de segmento de lista, escrevendo $lseg(\alpha, i, j)$ para representar o facto de a sequência α estar representada no segmento de lista que começa no endereço i e cujo o último nodo da sequência tem no campo seg o endereço j. A representação pode ser vista na Figura 3.6

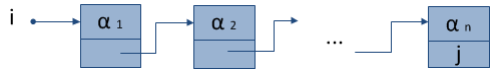


Figura 3.6: Segmento de uma lista ligada: $lseg(\alpha, i, j)$

Este predicado pode ser definido indutivamente por:

1. $lseg(\varepsilon, i, j) \stackrel{def}{=} emp \wedge i = j$

2. $lseg([a] \cdot \alpha, i, j) \stackrel{def}{=} \exists k. i \mapsto [valor : a, seg : k] * lseg(\alpha, k, j)$

Algumas propriedades sobre estes predicados:

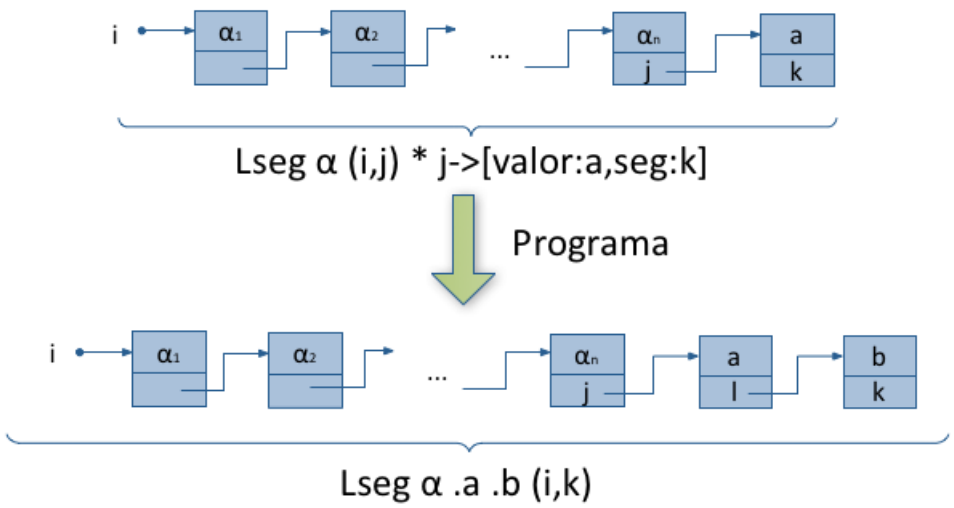
1. $lseg([a], i, j) \Leftrightarrow i \mapsto [valor : a, seg : j]$

2. $lseg(\alpha \cdot \beta, i, k) \Leftrightarrow \exists j. lseg(\alpha, i, j) * lseg(\beta, j, k)$

3. $lseg(\alpha \cdot [b], i, k) \Leftrightarrow \exists j. lseg(\alpha, i, j) * j \mapsto [valor : b, seg : k]$

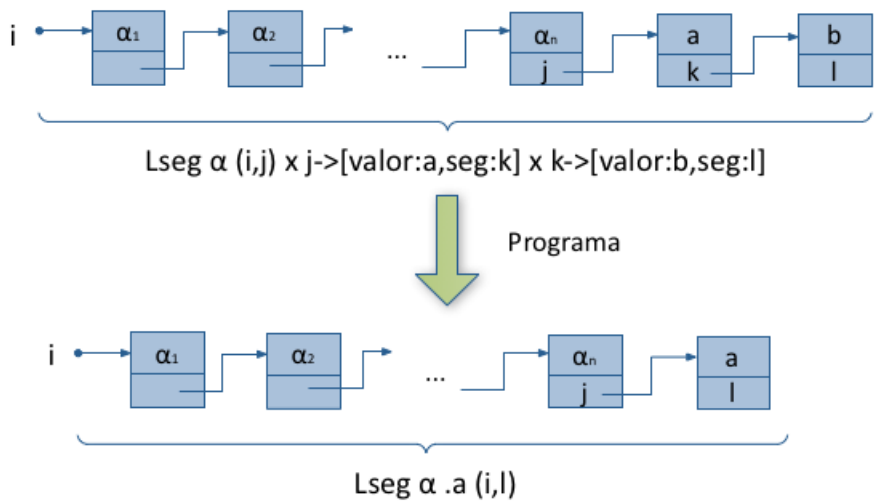
4. $lseg(\alpha, i, nil) \Leftrightarrow list(\alpha, i)$

Exemplo 3.4.2 Neste exemplo pretendemos ilustrar a introdução de uma célula no final da lista ligada. Além da apresentação do código será também ilustrado com a derivação do triplo que especifica o programa.
$\{lseg(\alpha, i, j) * j \mapsto [valor : a, seg : k]\}$
$\texttt{new}(l)$;
$l \rightarrow valor := b$;
$l \rightarrow seg := k$;
$j \rightarrow seg := l$;
$\{lseg(\alpha \cdot [a] \cdot [b], i, k)\}$



$\{lseg(\alpha,i,j) * j \mapsto [valor:a, seg:k]\}$
$\texttt{new}(l);$ Frame + Allocation
$\{lseg(\alpha,i,j) * j \mapsto [valor:a, seg:k] * l \mapsto [\,-\,]\}$
$l \rightarrow valor := b;$ Frame + Mutation
$\{lseg(\alpha,i,j) * j \mapsto [valor:a, seg:k] * l \mapsto [valor:b]\}$
$l \rightarrow seg := k;$ Frame + Mutation
$\{lseg(\alpha,i,j) * j \mapsto [valor:a, seg:k] * l \mapsto [valor:b, seg:k]\}$
$j \rightarrow seg := l;$ Frame + Mutation
$\{lseg(\alpha,i,j) * j \mapsto [valor:a, seg:l] * l \mapsto [valor:b, seg:k]\}$ (3)
$\{lseg(\alpha \cdot [a],i,l) * l \mapsto [valor:b, seg:k]\}$ (3)
$\{lseg(\alpha \cdot [a] \cdot [b],i,k)\}$

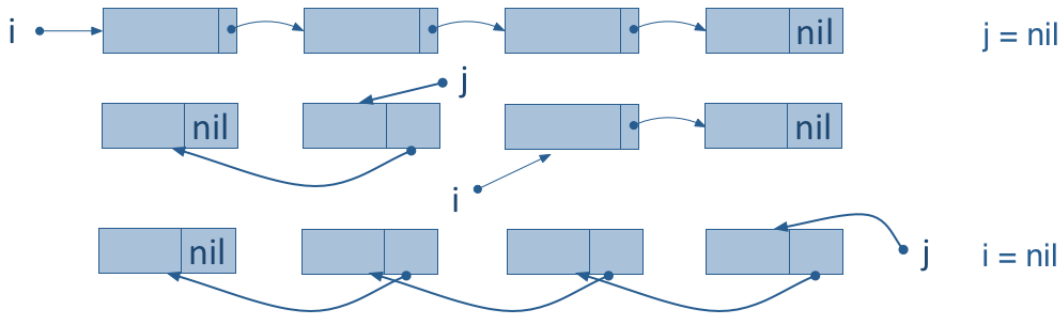
Exemplo 3.4.3 Neste exemplo pretendemos ilustrar a eliminação do elemento do final da lista, em tempo constante, sendo que j e k apontam para os últimos dois records.
$\{lseg(\alpha,i,j) * j \mapsto [valor:a, seg:k] * k \mapsto [valor:b, seg:l]\}$
$j \rightarrow seg := l;$
$\texttt{dispose}(k);$
$\{lseg(\alpha \cdot a,i,l)\}$



$$
\begin{array}{ll}
\{lseg(\alpha,i,j) * j \mapsto [valor:a, seg:k] * k \mapsto [valor:b, seg:l]\} & \\
j \rightarrow seg := l; & \text{Frame + Mutation} \\
\{lseg(\alpha,i,j) * j \mapsto [valor:a, seg:l] * k \mapsto [valor:b, seg:l]\} & \\
\texttt{dispose}(k); & \text{Frame + Dispose} \\
\{lseg(\alpha,i,j) * j \mapsto [valor:a, seg:l]\} & (3) \\
\{lseg(\alpha \cdot [a], i, l)\} &
\end{array}
$$

Exemplo 3.4.4 Neste exemplo pretendemos ilustrar o reverso de uma lista ligada. O programa que se segue inverte uma lista ligada. A lista original é apontada por i e a lista invertida começa no endereço j.

$$
\begin{array}{l}
\{list(\alpha, i)\} \\
j := nil; \\
while \ \neg \ (i \ = \ nil)\{ \\
\qquad k := i \rightarrow \ seg; \\
\qquad i \rightarrow \ seg := j; \\
\qquad j := i; \\
\qquad i := k; \\
\} \\
\{list(\alpha^+, j)\}
\end{array}
$$



$\{list(\alpha,i)\}$
$\{list(\alpha,i) * (emp \wedge nil = nil)\}$ Conseq.
$j := nil;$
$\{list(\alpha,i) * (emp \wedge j = nil)\}$ Def
$\{list(\alpha,i) * list(\varepsilon,j)\}$
$\{\exists\alpha_1,\beta(list(\alpha_1,i) * list(\beta,j)) \wedge \alpha^+ = \alpha_1^+.\beta\}$ Conseq.
INV:$\{\exists\alpha_1,\beta(list(\alpha_1,i) * list(\beta,j)) \wedge \alpha^+ = \alpha_1^+.\beta\}$
$while \neg(i = nil)\{$ Conseq. + While
$\{\exists a,\alpha_1,\beta(list([a]\cdot\alpha_1,i) * list(\beta,j)) \wedge \alpha^+ = (a.\alpha_1)^+.\beta\}$ Def
$\{\exists a,\alpha_1,\beta(i \mapsto [valor:a, seg:k] * list(\alpha_1,k) * list(\beta,j)) \wedge \alpha^+ = (a.\alpha_1)^+.\beta\}$
$k := i \rightarrow seg;$ Aux. Var. Elim. (3) + Frame + Conseq. + Lookup

$\{\exists a,\alpha_1,\beta(i \mapsto [valor:a, seg:k] * list(\alpha_1,k) * list(\beta,j)) \wedge \alpha^+ = (a.\alpha_1)^+.\beta\}$
$i \rightarrow seg := j;$ Aux. Var. Elim. (3) + Frame Conseq. + Lookup

$\{\exists a,\alpha_1,\beta(i \mapsto [valor:a, seg:j] * list(\alpha_1,k) * list(\beta,j)) \wedge \alpha^+ = (a.\alpha_1)^+.\beta\}$ Def
$\{\exists a,\alpha_1,\beta(list(\alpha_1,k) * list(a\cdot\beta,i)) \wedge \alpha^+ = \alpha_1^+.a.\beta\}$ Def
$\{\exists\alpha_1,\beta(list(\alpha_1,k) * list(\beta,i)) \wedge \alpha^+ = \alpha_1^+.\beta\}$
$j := i;$ Atrib.
$i := k;$ Atrib.
$\{\exists\alpha_1,\beta(list(\alpha_1,i) * list(\beta,j)) \wedge \alpha^+ = \alpha_1^+.\beta\}$
$\}$
$\{\exists\alpha_1,\beta(list(\alpha_1,i) * list(\beta,j)) \wedge \alpha^+ = \alpha_1^+.\beta\}$
$\{list(\alpha^+,j)\}$ Def

Capítulo 4

Verificação de programas $While_{pr}$

Neste capítulo, pretendemos estabelecer as bases para mecanizar a tarefa de verificação de programas $While_{pr}$. Dado um triplo de Hoare, o processo passará por usar as regras do sistema **S** no sentido *backwards*, identificando as instâncias de axiomas e respectivas condições laterais de forma a que fique garantida a validade de todas as condições laterais identificadas, estejamos, de facto, na posse de uma dedução do triplo no sistema **S**. As condições laterais identificadas são usualmente designadas por condições de verificação e esta fase do processo de verificação é designada por geração de condições de verificação.

Numa segunda fase do processo de verificação, deverá então ser analisado se as condições de verificação geradas são todas válidas (possivelmente esta tarefa é feita com recurso a *SMT Solvers* [4, 17]).

O nosso processo de geração de condições de verificação garante que, no caso de uma condição de verificação não ser válida, então não pode existir uma derivação do triplo de Hoare em causa e, portanto, o programa não respeitará a sua especificação.

Como motivação, consideremos de novo o triplo já analisado no capítulo anterior:

$\{x = 1 \wedge y = 2\}$
`new`(p);
$p \rightarrow valor := x$;
$x := y$;
$y := p \rightarrow valor$;
`dispose`(p)
$\{x = 2 \wedge y = 1\}$

Vimos já no capítulo anterior que existe uma derivação para este triplo. Mas será que $x = 1 \wedge y = 2$ é a pré-condição mais fraca para concluirmos $x = 2 \wedge y = 1$ após a execução do programa? Ou poderá existir uma pré-condição ainda mais fraca para este programa? Ao longo deste capítulo iremos preocupar-nos com este tipo de questão, bem como com a questão de garantir um conjunto de condições necessárias para um programa estar correcto.

Em resumo, o objectivo deste capítulo é mostrar que existe uma estratégia de prova tal que, se uma das condições laterais geradas não se verificar, então é porque não existe nenhuma derivação para o triplo de Hoare em causa. Esta estratégia dá origem à definição de um gerador de condições de verificação, usualmente chamado de **VCGen**.

4.1 $\mathbf{S^g}$: um sistema com regras globais

No sistema **S** para a lógica da separação que definimos no capítulo anterior, a maioria das regras apresenta uma propriedade importante: as fórmulas que ocorrem nas premissas da regra também ocorrem na conclusão da mesma. Um sistema em que todas as regras satisfaçam esta propriedade é possível ser mecanizado, não sendo necessário "adivinhar" condições intermédias para aplicar as regras. A excepção a esta regra é a regra da composição.

$$\frac{\{\phi\}\mathsf{C}_1\{\theta\} \qquad \{\theta\}\mathsf{C}_2\{\psi\}}{\{\phi\}\mathsf{C}_1;\mathsf{C}_2\{\psi\}}$$

Note-se que nesta regra, θ ocorre nas premissas sem que ocorra na conclusão.

Uma segunda propriedade desejável é a ausência de ambiguidade na escolha da regra a aplicar. Note-se que o sistema **S** não têm esta propriedade. As regras de *frame*, consequência e eliminação de variáveis auxiliares podem ser aplicadas com um comando C arbitrário e, por isso, podem produzir os mesmos triplos produzidos por outras regras.

Ao longo deste capítulo iremos considerar um sistema dedutivo onde estas regras deixarão de ser primitivas, mas serão admissíveis, e portanto continuaremos a poder deduzir os mesmos triplos de *Hoare*. Para assegurarmos a admissibilidade das regras será necessário promover alterações nas restantes regras.

Começaremos por considerar um sistema (que representaremos por $\mathbf{S^g}$ - g de global) onde as regras de *frame* e de eliminação de variáveis auxiliares deixarão de ser permitidas, e depois um sistema (que notaremos, por $\mathbf{S^{gb}}$ - gb de *global backwards*) em que, adicionalmente, eliminaremos também a regra da consequência. Em particular, a eliminação da regra de *frame* tem impacto nas diversas regras relativas aos comandos que actuam sobre a *heap*.

A remoção da regra de eliminação de variáveis auxiliares apenas interfere com a regra do *lookup*, por esta ser a única em que na pré/pós-condição há variáveis que não são totalmente determinadas pelo comando.

A eliminação da regra da consequência tem impacto sobre a maioria das outras regras, permitindo a introdução de uma pré-condição mais forte, desde que esta implique a pré-condição mais fraca para a regra.

As regras do sistema $\mathbf{S^g}$ encontram-se na Figura 4.1. Em comparação com as regras de **S**, é de salientar que nos axiomas relativos aos comandos que manipulam a

heap há agora uma pré e pós-condição com uma conjunção separada (à semelhança da conclusão da regra de *frame* do sistema **S**).

De salientar ainda que, na regra do *lookup*, v_1 na pré-condição e v_2 na pós-condição passam a estar quantificados.

$$\frac{}{\{\phi\}\texttt{skip}\{\phi\}}\ \text{Skip}$$

$$\frac{}{\{\psi[a/v]\}v{:=}a\{\psi\}}\ \text{Atrib.}$$

$$\frac{\{\phi\}\mathsf{C}_1\{\theta\} \qquad \{\theta\}\mathsf{C}_2\{\psi\}}{\{\phi\}\mathsf{C}_1;\mathsf{C}_2\{\psi\}}\ \text{Comp.}$$

$$\frac{\{\phi \wedge b\}\mathsf{C}_t\{\psi\} \qquad \{\phi \wedge \neg b\}\mathsf{C}_f\{\psi\}}{\{\phi\}\texttt{if}\ b\ \texttt{then}\ \mathsf{C}_t\ \texttt{else}\ \mathsf{C}_f\{\psi\}}\ \text{If}$$

$$\frac{\{\theta \wedge b\}\mathsf{C}\{\theta\}}{\{\theta\}\texttt{while}\ b\ \{\mathsf{C}\}\{\theta \wedge \neg b\}}\ \text{While}$$

$$\frac{\models \phi' \Rightarrow \phi \qquad \{\phi\}\mathsf{C}\{\psi\} \qquad \models \psi \Rightarrow \psi'}{\{\phi'\}\mathsf{C}\{\psi'\}}\ \text{Conseq.}$$

$$\frac{}{\{p \mapsto [f:-] * \theta\}p{\rightarrow}f{:=}a\{p \mapsto [f:a] * \theta\}}\ \text{Mutation}$$

$$\frac{}{\{p \mapsto [-] * \theta\}\texttt{dispose}(p)\{\theta\}}\ \text{Dispose}$$

$$\frac{p \notin free(\theta)}{\{\theta\}\texttt{new}(p)\{p \mapsto [-] * \theta\}}\ \text{Allocation}$$

$$\frac{v \notin free(\theta) \qquad v_1 \neq v_2 \wedge v \neq v_1 \wedge v \neq v_2 \wedge v_1 \neq p \wedge v_2 \neq p}{\{\exists v_1.(p \mapsto [f:v_1] * (\theta[v/v_2]))\}v{:=}p \rightarrow f\{\exists v_2.((p[v_2/v] \mapsto [f:v]) * (\theta[v/v_1]))\}}\ \text{Lookup}$$

Figura 4.1: $\mathbf{S^g}$- Sistema sem as regras de *Frame* e da eliminação de variáveis auxiliares

Serão de seguida apresentados os teoremas da equivalência entre o sistema **S** e o sistema $\mathbf{S^g}$ e respectivas demonstrações. Com este resultado provado, fica também garantida a correcção do sistema $\mathbf{S^g}$ relativamente à semântica. Este resultado ser-nos-á muito útil mais adiante. De salientar que, neste capítulo, apenas serão apresentados os casos da prova relativos às regras de manipulação de memória dinâmica, devido à dimensão da prova. Os casos relativos à linguagem *While* simples não serão apresentados, uma vez que as demonstrações são rotineiras. Por exemplo: aplica-se exactamente o mesmo axioma no caso do *skip* e da atribuição e a correspondente

hipótese de indução com a correspondente regra, nos casos da composição, *if* e *while*.

Teorema 4.1.1 Se $\vdash_{S^g} \{\phi\}\mathsf{C}\{\psi\}$, então $\vdash_S \{\phi\}\mathsf{C}\{\psi\}$.

Demonstração Por indução nas deduções de $\vdash_{S^g} \{\phi\}\mathsf{C}\{\psi\}$. Provaremos que se tivermos uma dedução em $\mathbf{S^g}$ de $\{\phi\}\mathsf{C}\{\psi\}$, então existe uma dedução em $\mathbf{S}$ deste triplo.

1. Caso *mutation*:

$$\frac{}{\{p \mapsto [f:-] * \theta\}p{\rightarrow}f{:=}a\{p \mapsto [f:a] * \theta\}}\ \text{Mutation}$$

   Queremos demonstrar que $\vdash_S \{p \mapsto [f:-] * \theta\}p{\rightarrow}f{:=}a\{p \mapsto [f:a] * \theta\}$.

   Para tal, basta construir a seguinte dedução em $\mathbf{S}$:

$$\dfrac{\dfrac{}{\{p \mapsto [f:-]\}p{\rightarrow}f{:=}a\{p \mapsto [f:a]\}}\ \text{Mutation}}{\{p \mapsto [f:-] * \theta\}p{\rightarrow}f{:=}a\{p \mapsto [f:a] * \theta\}}\ \text{Frame}$$

   Note-se que, de facto, $modifies(p{\rightarrow}f{:=}a) \cap free(\theta) = \varnothing$ (necessário à aplicação da Frame), dado que $modifies(p{\rightarrow}f{:=}a) = \varnothing$

2. Caso *dispose*:

$$\frac{}{\{p \mapsto [-] * \theta\}\texttt{dispose}(p)\{\theta\}}\ \text{Dispose}$$

   Queremos demonstrar que $\vdash_S \{p \mapsto [-] * \theta\}\texttt{dispose}(p)\{\theta\}$.

   Ora, basta então tomar,

$$\dfrac{\dfrac{}{\{p \mapsto [-]\}\texttt{dispose}(p)\{emp\}}\ \text{Dispose}}{\{p \mapsto [-] * \theta\}\texttt{dispose}(p)\{\theta\}}\ \text{Frame}$$

   e note-se que, de facto, $modifies(\texttt{dispose}(p)) \cap free(\theta) = \varnothing$, dado que $modifies(\texttt{dispose}(p)) = \varnothing$

3. Caso *allocation*:

$$\frac{p \notin free(\theta)}{\{\theta\}\texttt{new}(p)\{p \mapsto [-] * \theta\}}\ \text{Allocation}$$

   Queremos demonstrar que $\vdash_S \{\theta\}\texttt{new}(p)\{p \mapsto [-] * \theta\}$.

   Basta então tomar, a dedução em $\mathbf{S}$:

$$\dfrac{\dfrac{\dfrac{}{\{emp\}\texttt{new}(p)\{p \mapsto [-]\}}\ \text{Allocation}}{\{emp * \theta\}\texttt{new}(p)\{p \mapsto [-] * \theta\}}\ \text{Frame}}{\{\theta\}\texttt{new}(p)\{p \mapsto [-] * \theta\}}\ \text{Conseq.}$$

Observa-se de novo que a condição lateral necessária a aplicação da regra de *frame* é satisfeita, pois $modifies(\texttt{new}(p)) \cap free(\theta) = \emptyset \Leftrightarrow p \notin free(\theta)$, e esta última condição é um pressuposto da regra *allocation* em $\mathbf{S^g}$. Na aplicação da regra da consequência usa-se a equivalência $emp * \theta \Leftrightarrow \theta$.

4. Caso *lookup*:

$$\dfrac{v \notin free(\theta) \wedge v_1 \neq v_2 \wedge v_1 \neq v \wedge v_2 \neq v \wedge v_1 \neq p \wedge v_2 \neq p}{\{\exists v_1.(p \mapsto [f : v_1] * (\theta[v/v_2]))\}v{:=}p \rightarrow f\{\exists v_2.((p[v_2/v] \mapsto [f : v]) * (\theta[v/v_1]))\}}\ \text{Lookup}$$

Queremos demonstrar que
$\vdash_S \{\exists v_1.(p \mapsto [f : v_1] * (\theta[v/v_2]))\}v{:=}p \rightarrow f\{\exists v_2.((p[v_2/v] \mapsto [f : v]) * (\theta[v/v_1]))\}$.
Ora, considerando,

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{v \notin \mathbf{V}ar(v_1) \cup \mathbf{V}ar(v_2)}{\{v = v_2 \wedge p \mapsto [f : v_1]\}v{:=}p \rightarrow f\{v = v_1 \wedge p[v_2/v] \mapsto [f : v_1]\}}\ \text{Lookup}}{\{(v = v_2 \wedge p \mapsto [f : v_1]) * \theta\}v{:=}p \rightarrow f\{(v = v_1 \wedge p[v_2/v] \mapsto [f : v_1]) * \theta\}}\ \text{Frame}}{\{(v = v_2 \wedge p \mapsto [f : v_1]) * \theta[v/v_2]\}v{:=}p \rightarrow f\{(v = v_1 \wedge p[v_2/v] \mapsto [f : v_1]) * \theta[v/v_1]\}}\ \text{Conseq.2}}{\{\exists v_1.\exists v_2.((v = v_2 \wedge p \mapsto [f : v_1]) * \theta[v/v_2])\}v{:=}p \rightarrow f\{\exists v_1.\exists v_2.((v = v_1 \wedge p[v_2/v] \mapsto [f : v_1]) * \theta[v/v_1])\}}\ \text{Aux. Var. Elim.}(2\textit{vezes})}{\{\exists v_1.(p \mapsto [f : v_1] * (\theta[v/v_2]))\}v{:=}p \rightarrow f\{\exists v_2.((p[v_2/v] \mapsto [f : v]) * (\theta[v/v_1]))\}}\ \text{Conseq.1}$$

Note-se que:

(a) Conseq. 1 - As condições laterais necessárias seguem das equivalências lógicas $\phi[v/v'] \leftrightarrow \exists v'(v = v' \wedge \phi[v/v'])$ e $(m = a \wedge \phi) * \psi \Leftrightarrow m = a \wedge (\phi * \psi)$, sendo que a condição relativa às pós-condição requer ainda a equivalência $\exists v_1(v_2\phi) \Leftrightarrow \exists v_2(\exists v_1\phi)$.

(b) Aux. Var. Elim. - Da suposição $v_1 \neq v \wedge v_2 \neq v \wedge v_1 \neq p \wedge v_2 \neq p$, $v_1, v_2 \notin free(v{:=}p \rightarrow f)$, e podemos aplicar correctamente a regra de eliminação de variáveis auxiliares.

(c) Conseq. 2- As condições laterais seguem das equivalências lógicas $(m = a \wedge \phi) * \psi \Leftrightarrow \phi * (m = a \wedge \psi)$ e $v = v' \wedge \phi \Leftrightarrow v = v' \wedge \phi[v/v']$ e ainda da propriedade $\phi \Leftrightarrow \psi \Rightarrow \gamma * \phi \Leftrightarrow \gamma * \psi$.

(d) Frame - Para aplicar a regra *frame* temos que garantir que $modifies(v{:=}p \rightarrow f) \cup free(\theta) = \emptyset$, o que decorre da suposição $v \notin free(\theta)$.

(e) Lookup - $v \notin \mathbf{V}ar(v_1) \cup \mathbf{V}ar(v_2)$ decorre de termos suposto que $v \neq v_1$ e $v \neq v_2$.

□

Teorema 4.1.2 Se $\vdash_S \{\phi\}\mathsf{C}\{\psi\}$, então $\vdash_{S^g} \{\phi\}\mathsf{C}\{\psi\}$.

Demonstração Por indução na deduções de $\vdash_S \{\phi\}\mathsf{C}\{\psi\}$.

1. Caso *mutation*:

$$\frac{}{\{p \mapsto [f:-]\}p{\rightarrow}f{:=}a\{p \mapsto [f:a]\}}\ \text{Mutation}$$

   Queremos demonstrar que $\vdash_{S^g} \{p \mapsto [f:-]\}p{\rightarrow}f{:=}a\{p \mapsto [f:a]\}$. Basta então tomar,

$$\dfrac{\dfrac{}{\{p \mapsto [f:-] * emp\}p{\rightarrow}f{:=}a\{p \mapsto [f:a] * emp\}}\ \text{Mutation}}{\{p \mapsto [f:-]\}p{\rightarrow}f{:=}a\{p \mapsto [f:a]\}}\ \text{Conseq.}$$

   Falta provar as condições laterais necessárias à aplicação da regra da consequência, ou seja, $\models p \mapsto [f:-] \Rightarrow p \mapsto [f:-] * emp$ e $\models p \mapsto [f:a] * emp \Rightarrow p \mapsto [f:a]$, mas ambas são válidas e seguem da equivalência $\theta * emp \Leftrightarrow \theta$ que é válida para qualquer θ.

2. Caso *dispose*:

$$\frac{}{\{p \mapsto [-]\}\texttt{dispose}(p)\{emp\}}\ \text{Dispose}$$

   Queremos demonstrar que$\vdash_{S^g} \{p \mapsto [-]\}\texttt{dispose}(p)\{emp\}$.

   Basta então tomar,

$$\dfrac{\dfrac{}{\{p \mapsto [-] * emp\}\texttt{dispose}(p)\{emp\}}\ \text{Dispose}}{\{p \mapsto [-]\}\texttt{dispose}(p)\{emp\}}\ \text{Conseq.}$$

   Novamente, a condição necessária à aplicação da regra da consequência segue de $\theta * emp \Leftrightarrow \theta$ (para todo o θ).

3. Caso *allocation*:

$$\frac{}{\{emp\}\texttt{new}(p)\{p \mapsto [-]\}}\ \text{Allocation}$$

   Queremos demonstrar que $\vdash_{S^g} \{emp\}\texttt{new}(p)\{p \mapsto [-]\}$.

   Basta então tomar,

$$\dfrac{\dfrac{}{\{emp\}\texttt{new}(p)\{p \mapsto [-] * emp\}}\ \text{Allocation}}{\{emp\}\texttt{new}(p)\{p \mapsto [-]\}}\ \text{Conseq.}$$

Mais uma vez, na aplicação da regra da consequência é usada a equivalência $\theta * emp \Leftrightarrow \theta$.

4. Caso *lookup*

$$\frac{v \notin Vars(a) \cup Vars(m)}{\{v = m \wedge p \mapsto [f : a]\}v{:=}p \rightarrow f\{v = a \wedge p[m/v] \mapsto [f : a]\}} \text{ Lookup}$$

Queremos demonstrar que $\vdash_{Sg} \{v = m \wedge p \mapsto [f : a]\}v{:=}p \rightarrow f\{v = a \wedge p[m/v] \mapsto [f : a]\}$.

Ora, tomando variáveis frescas v' e v'', podemos construir a derivação,

$$\dfrac{\dfrac{\dfrac{\dfrac{v \notin free(v' = m \wedge v'' = a \wedge emp) \qquad v \neq v' \wedge v \neq v'' \wedge v' \neq v'' \wedge v' \neq p \wedge v'' \neq p}{\{\exists .v''.p \mapsto [f : v''] * ((v' = m \wedge v'' = a \wedge emp)[v/v'])\}v{:=}p \rightarrow f\{\exists v'.(p[v'/v] \mapsto [f : v]) * ((v' = m \wedge v'' = a \wedge emp)[v/v''])\}} \text{ Lookup}}{\{\exists v''.p \mapsto [f : v''] * (v = m \wedge v'' = a \wedge emp)\}v{:=}p \rightarrow f\{\exists v'.p[v'/v] \mapsto [f : v] * (v' = m \wedge v = a \wedge emp)\}} \text{ Igualdade}}{\{\exists v''.v = m \wedge v'' = a \wedge p \mapsto [f : v'']\}v{:=}p \rightarrow f\{\exists v'.v' = m \wedge v = a \wedge p[v'/v] \mapsto [f : v]\}} \text{ Conseq. 2}}{\{v = m \wedge p \mapsto [f : a]\}v{:=}p \rightarrow f\{v = a \wedge p[m/v] \mapsto [f : a]\}} \text{ Conseq. 1}$$

Note-se que:

(a) o passo identificado por "Igualdade" não compreende uma inferência em $\mathbf{S^g}$, na verdade, os triplos de Hoare acima e a baixo do traço são iguais o triplo abaixo resulta do triplo acima por execução da substituição.

(b) Conseq. 1: A prova das condições laterais encontra-se em apêndice nos lemas A.1.3 e A.1.4.

(c) Conseq. 2: A prova das condições laterais encontra-se em apêndice nos lemas A.1.5 e A.1.6.

(d) Lookup: a satisfação das duas condições laterais seguem da suposição $v \notin Vars(a) \cup Vars(m)$ e do facto de v' e v'' serem variáveis frescas.

Os casos relativos a regra de *Frame* e da regra de eliminação de variáveis auxiliares resultam da aplicação da hipótese de indução e da admissibilidade destas duas regras no sistema $\mathbf{S^g}$ (Proposições 4.1.3 e 4.1.4 abaixo).

□

Tal como foi dito anteriormente, apesar da eliminação da regra de *Frame* e da regra de eliminação de variáveis auxiliares deste sistema, estas regras continuam a ser admissíveis. Serão agora enunciados as proposições e elaboradas as demonstrações que estabelecem estas admissibilidades.

Proposição 4.1.3 Se $\vdash_{Sg} \{\phi\}\mathsf{C}\{\psi\}$ e $modifies(\mathsf{C}) \cap free(\theta) = \emptyset$, então $\vdash_{Sg} \{\phi * \theta\}\mathsf{C}\{\psi * \theta\}$.

Demonstração Por indução na deduções de $\vdash_{Sg} \{\phi\}\mathsf{C}\{\psi\}$:

1. Caso *mutation*:

$$\frac{}{\{p \mapsto [f : -] * \theta_1\}p{\rightarrow}f{:=}a\{p \mapsto [f : a] * \theta_1\}}\ \text{Mutation}$$

Queremos demonstrar que se $modifies(p{\rightarrow}f{:=}a) \cap free(\theta) = \varnothing$ então $\vdash_{Sg} \{(p \mapsto [f : -] * \theta_1) * \theta\}p{\rightarrow}f{:=}a\{(p \mapsto [f : a] * \theta_1) * \theta\}$.

Basta então tomar,

$$\dfrac{\dfrac{}{\{p \mapsto [f : -] * (\theta_1 * \theta)\}p{\rightarrow}f{:=}a\{p \mapsto [f : a] * (\theta_1 * \theta)\}}\ \text{Mutation}}{\{(p \mapsto [f : -] * \theta_1) * \theta\}p{\rightarrow}f{:=}a\{(p \mapsto [f : a] * \theta_1) * \theta\}}\ \text{Conseq.}$$

A validade das condições laterais necessária à aplicação da regra da consequência é garantida pela associatividade da conjunção separada (Proposição 3.1.7).

2. Caso *dispose*:

$$\frac{}{\{p \mapsto [\ -\ ] * \theta_1\}\texttt{dispose}(p)\{\theta_1\}}\ \text{Dispose}$$

Queremos demonstrar que se $modifies(\texttt{dispose}(p)) \cap free(\theta) = \varnothing$ então $\vdash_{Sg}$ $\{(p \mapsto [\ -\ ] * \theta_1) * \theta\}\texttt{dispose}(p)\{(emp * \theta_1) * \theta\}$.

Basta então tomar,

$$\dfrac{\dfrac{}{\{p \mapsto [\ -\ ] * (\theta_1 * \theta)\}\texttt{dispose}(p)\{\theta_1 * \theta\}}\ \text{Dispose}}{\{(p \mapsto [\ -\ ] * \theta_1) * \theta\}\texttt{dispose}(p)\{\theta_1) * \theta\}}\ \text{Conseq.}$$

Note-se que as condições laterais necessárias à aplicação da regra da consequência segue novamente pela associatividade da conjunção separada (Proposição 3.1.7).

3. Caso *allocation*:

$$\frac{p \notin free(\theta_1)}{\{\theta_1\}\texttt{new}(p)\{p \mapsto [-] * \theta_1\}}\ \text{Allocation}$$

Queremos demonstrar que se $modifies(\texttt{new}(p)) \cap free(\theta) = \varnothing$ então $\vdash_{Sg} \{\theta_1 * \theta\}\texttt{new}(p)\{(p \mapsto [-] * \theta_1) * \theta\}$.

Ora, basta então tomar,

$$\dfrac{\dfrac{p \notin free(\theta_1 * \theta)}{\{\theta_1 * \theta\}\texttt{new}(p)\{p \mapsto [-] * (\theta_1 * \theta)\}}\ \text{Allocation}}{\{\theta_1 * \theta\}\texttt{new}(p)\{(p \mapsto [-] * \theta_1) * \theta\}}\ \text{Conseq.}$$

Note-se que $p \notin free(\theta_1 * \theta)$ segue da suposição inicial $p \notin free(\theta_1)$ e da suposição $modifies(\texttt{new}(p)) \cap free(\theta) = \emptyset$, que implicam que $p \notin free(\theta)$. Para aplicação da regra da consequência usamos de novo a associatividade da conjunção separada (Proposição 3.1.7).

4. Caso *lookup*:

$$\frac{v \notin free(\theta_1) \wedge v_1 \neq v_2 \wedge v_1 \neq v \wedge v_2 \neq v \wedge v_1 \neq p \wedge v_2 \neq p}{\{\exists v_1.p \mapsto [f : v_1] * \theta_1[v/v_2]\}v{:=}p \to f\{\exists v_2.(p[v_2/v] \mapsto [f : v]) * \theta_1[v/v_1]\}} \text{ Lookup}$$

Queremos demonstrar que se $modifies(v{:=}p \to f) \cap free(\theta) = \emptyset$ então $\vdash_{Sg} \{(\exists v_1.p \mapsto [f : v_1] * \theta_1[v/v_2]) * \theta\}v{:=}p \to f\{(\exists v_2.p[v_2/v] \mapsto [f : v] * \theta_1[v/v_1]) * \theta\}$.

Ora, basta então tomar,

$$\dfrac{\dfrac{v \notin free(\theta * \theta_1) \qquad v_1 \neq v_2 \wedge v \neq v_1 \wedge v \neq v_2 \wedge v_1 \neq p \wedge v_2 \neq p}{\{\exists v_1.p \mapsto [f : v_1] * (\theta_1 * \theta[v/v_2])\}v{:=}p \to f\{\exists v_2.p[v_2/v] \mapsto [f : v] * (\theta_1 * \theta[v/v_1])\}} \text{ Lookup}}{\{(\exists v_1.p \mapsto [f : v_1] * \theta_1[v/v_2]) * \theta\}v{:=}p \to f\{(\exists v_2.p[v_2/v] \mapsto [f : v] * \theta_1[v/v_1]) * \theta\}} \text{ Conseq.}$$

   (a) Conseq. - As condições laterais da regra da consequência por associatividade da conjunção separada (Proposição 3.1.7).

   (b) Lookup - As condições da regra *lookup* são $v \notin free(\theta * \theta_1)$ e $v_1 \neq v_2 \wedge v \neq v_1 \wedge v \neq v_2 \wedge v_1 \neq p \wedge v_2 \neq p$. A primeira segue de $v \notin free(\theta)$ e $modifies(v{:=}p \to f) \cap free(\theta_1) = \emptyset$. A segunda foi suposta inicialmente.

5. Caso composição:

$$\frac{\dfrac{D_1}{\{\phi\}\mathsf{C}_1\{\gamma\}} \qquad \dfrac{D_2}{\{\gamma\}\mathsf{C}_2\{\psi\}}}{\{\phi\}\mathsf{C}_1;\mathsf{C}_2\{\psi\}} \text{ Comp.}$$

Queremos demonstrar que se $modifies(\mathsf{C}_1;\mathsf{C}_2) \cap free(\theta) = \emptyset$ então $\vdash_{Sg} \{\phi * \theta\}\mathsf{C}_1;\mathsf{C}_2\{\psi * \theta\}$.

Da suposição sai que $modifies(\mathsf{C}_1) \cap free(\theta) = \emptyset$ e $modifies(\mathsf{C}_2) \cap free(\theta) = \emptyset$ e aplicando as hipóteses de indução associadas a D_1 e D_2, ou seja, que existem derivações D_1' e D_2' de $\{\phi * \theta\}\mathsf{C}_1\{\gamma * \theta\}$ e de $\{\gamma * \theta\}\mathsf{C}_2\{\psi * \theta\}$. Assim, basta então tomar,

$$\frac{\dfrac{D_1'}{\{\phi * \theta\}\mathsf{C}_1\{\gamma * \theta\}} \qquad \dfrac{D_2'}{\{\gamma * \theta\}\mathsf{C}_2\{\psi * \theta\}}}{\vdash_{Sg} \{\phi * \theta\}\mathsf{C}_1;\mathsf{C}_2\{\psi * \theta\}} \text{ Comp.}$$

Os restantes casos relativos à linguagem *While* simples estão apresentados em anexo na Proposição A.1.18.

□

Proposição 4.1.4 Se $\vdash_{Sg} \{\phi\}\mathsf{C}\{\psi\}$ e $v \notin free(\mathsf{C})$, então $\vdash_{Sg} \{\exists v.\phi\}\mathsf{C}\{\exists v.\psi\}$

Demonstração Por indução na derivação de $\vdash_{Sg} \{\phi\}\mathsf{C}\{\psi\}$:

1. Caso *mutation*:

$$\frac{}{\{p \mapsto [f:-] * \theta\}p{\rightarrow}f{:=}a\{p \mapsto [f:a] * \theta\}}\ \text{Mutation}$$

   Suponhamos que $v \notin free(p{\rightarrow}f{:=}a)$. Queremos provar $\vdash_{Sg} \{\exists v.(p \mapsto [\,-\,] * \theta)\}p{\rightarrow}f{:=}a\{\exists v.(p \mapsto [f:a] * \theta)\}$.

   Basta então tomar,

$$\frac{\dfrac{}{\{p \mapsto [\,-\,] * (\exists v.\theta)\}p{\rightarrow}f{:=}a\{p \mapsto [f:a] * (\exists v.\theta)\}}\ \text{Mutation}}{\{\exists v.(p \mapsto [\,-\,] * \theta)\}p{\rightarrow}f{:=}a\{\exists v.(p \mapsto [f:a] * \theta)\}}\ \text{Conseq.}$$

   Falta apenas provar as condições laterais necessárias à aplicação da consequência, isto é, que $\models \exists v.p \mapsto [\,-\,] * \theta \Rightarrow p \mapsto [\,-\,] * (\exists v.\theta)$ e $\models v \mapsto [f:a] * (\exists v.\theta) \Rightarrow \exists v.p \mapsto [f:a] * \theta$.

   A validade de ambas segue do facto de $v \notin free(p{\rightarrow}f{:=}a)$ e portanto $v \neq p$ e $v \notin Vars(a)$.

2. Caso *dispose*:

$$\frac{}{\{p \mapsto [-] * \theta\}\texttt{dispose}(p)\{\theta\}}\ \text{Dispose}$$

   Queremos demonstrar que se então $\vdash_{Sg} \{\exists v.(p \mapsto [\,-\,] * \theta)\}\texttt{dispose}(p)\{\exists v.\theta\}$. Basta então tomar,

$$\frac{\dfrac{}{\{p \mapsto [\,-\,] * (\exists v.\theta)\}\texttt{dispose}(p)\{\exists v.\theta\}}\ \text{Dispose}}{\{\exists v.p \mapsto [\,-\,] * \theta\}\texttt{dispose}(p)\{\exists v.\theta\}}\ \text{Conseq.}$$

   As condições necessárias à aplicação da consequência, designadamente $\models \exists v.p \mapsto [\,-\,] * \theta \Rightarrow p \mapsto [\,-\,] * (\exists v.\theta)$ e $\models (\exists v.\theta) \Rightarrow \exists v.\theta$ são válidas. A segunda é imediata. A primeira de $p \neq v$ a "descrição da heap" pode passar quer para dentro quer para fora da quantificação existencial.

3. Caso *allocation*:

$$\frac{p \notin free(\theta)}{\{\theta\}\texttt{new}(p)\{p \mapsto [-] * \theta\}}\ \text{Allocation}$$

Queremos demonstrar que se $v \notin free(\texttt{new}(p))$ então $\vdash_{Sg} \{\exists v.\theta\}\texttt{new}(p)\{\exists v.(p \mapsto [\,-\,] * \theta)\}$.

Basta então tomar,

$$\frac{\dfrac{}{\{\exists v.\theta\}\texttt{new}(p)\{p \mapsto [\,-\,] * (\exists v.\theta)\}}\ \text{Allocation}}{\{\exists v.\theta\}\texttt{new}(p)\{\exists v.(p \mapsto [\,-\,] * \theta\})}\ \text{Conseq.}$$

A nível das condições laterais é necessario que $p \notin free(\exists v.\theta)$, $\models \exists v.\theta \Rightarrow \exists v.\theta$ e ainda $\models p \mapsto [\,-\,] * (\exists v.\theta) \Rightarrow \exists v.(p \mapsto [\,-\,] * \theta)$.

Da suposição que $p \notin free(\theta)$ segue que $p \notin free(\exists v.\theta)$.

Como $v \neq p$, a "descrição da heap" pode passar quer para dentro quer para fora da quantificação existencial.

4. Caso *lookup*:

$$\frac{v' \notin free(\theta) \qquad v_1 \neq v_2 \wedge v' \neq v_1 \wedge v' \neq v_2 \wedge v_1 \neq p \wedge v_2 \neq p}{\{\exists v_1.p \mapsto [f : v_1] * \theta[v'/v_2]\}v' := p \to f\{\exists v_2.p[v_2/v'] \mapsto [f : v'] * \theta[v'/v_1]\}}\ \text{Lookup}$$

Queremos demonstrar que se $v \notin free(v' := p \to f)$ então

$\vdash_{Sg} \{\exists v.(\exists v_2.p \mapsto [f : v_1] * \theta[v'/v_1])\}v{:=}p \to f\{\exists v.(\exists v_1.(p[v_2/v'] \mapsto [f : v'] * \theta[v'/v_1])\}$.

Basta então tomar,

$$\frac{\dfrac{v' \notin free(\exists v.\theta) \qquad v_1 \neq v_2 \wedge v' \neq v_1 \wedge v' \neq v_2 \wedge v_1 \neq p \wedge v_2 \neq p}{\{\exists v_1.(p \mapsto [f : v_1]) * (\exists v.\theta)[v'/v_2]\}v{:=}p \to f\{\exists v_2.(p[v_2/v'] \mapsto [f : v']) * (\exists v.\theta)[v'/v_1]\}}\ \text{Lookup}}{\{\exists v.\exists v_1.p \mapsto [f : v_1] * \theta[v'/v_2]\}v{:=}p \to f\{\exists v.\exists v_1.(p[v_2/v'] \mapsto [f : v'] * \theta[v'/v_1]\}}\ \text{Conseq.}$$

Para podermos aplicar a regra da consequência, temos que demonstrar $\models \exists v.\exists v_1.p \mapsto [f : v_1] * \theta[v/v_2] \Rightarrow \exists v_1.(p \mapsto [f : v_1]) * (\exists v.\theta)[v'/v_2]$ e $\models \exists v_2.(p[v_2/v'] \mapsto [f : v']) * (\exists v.\theta)[v'/v_1] \Rightarrow \exists v.\exists v_1.(p[v_2/v'] \mapsto [f : v'] * \theta[v'/v_1]$, mas de sabermos que $v \notin free(v' := p \to f)$, temos em particular que $v \neq v' \wedge v \neq p$, logo a "descrição da heap" pode passar quer para dentro quer para fora da quantificação existencial.

Para podermos aplicar a regra *Lookup* temos que garantir $v_1 \neq v_2 \wedge v' \neq v_1 \wedge v' \neq v_2 \wedge v_1 \neq p \wedge v_2 \neq p$ que seguem das suposições relativos ao caso da regra *lookup*.

5. Caso composição

$$\frac{\dfrac{D_1}{\{\phi\}\mathsf{C}_1\{\gamma\}} \quad \dfrac{D_2}{\{\gamma\}\mathsf{C}_2\{\psi\}}}{\{\phi\}\mathsf{C}_1;\mathsf{C}_2\{\psi\}}\ \text{Comp.}$$

Queremos demonstrar que se $modifies(\mathsf{C}_1;\mathsf{C}_2)\cap free(\theta)=\{\}$ então $\vdash_{S^{gb}} \{\exists v.\phi\}\mathsf{C}_1;\mathsf{C}_2\{\exists v.\psi\}$.

De sabermos que $v \notin free(\mathsf{C}_1;\mathsf{C}_2)$ temos em particular que $v \notin free(\mathsf{C}_1)$ e $v \notin free(\mathsf{C}_2)$. Logo, as hipóteses de indução associadas a D_1 e D_2 permitem concluir que existem as derivações D_1' e D_2' de $\{\exists v.\phi\}\mathsf{C}_1\{\exists v.\theta\}$ e de $\{\exists v.\theta\}\mathsf{C}_2\{\exists v.\psi\}$.

Como tal, basta tomar a derivação:

$$\frac{\dfrac{D_1'}{\{\exists v.\phi\}\mathsf{C}_1\{\exists v.\theta\}} \quad \dfrac{D_2'}{\{\exists v.\theta\}\mathsf{C}_2\{\exists v.\psi\}}}{\{\exists v.\phi\}\mathsf{C}_1;\mathsf{C}_2\{\exists v.\psi\}}\ \text{Comp.}$$

Os restantes casos relativos à linguagem *While* simples serão apresentados em anexo na Proposição A.1.19.

□

4.2 S^{gb}: um sistema *goal directed*

Depois de eliminadas as regras da *frame* e eliminação de variáveis auxiliares, resta agora eliminar a regra da consequência para termos um sistema *goal-directed*, ou seja, um sistema onde o triplo de *Hoare* é no máximo conclusão de uma regra (a regra associada ao comando presente no triplo). Para garantirmos a admissibilidade da regra da consequência, a estratégia passa por distribuir a regra da consequência pelas várias regras do sistema de inferência. É de salientar que para comandos usais relativos à linguagem *While* simples as regras deste sistema correspondem às regras apresentadas em [3]. As novas regras, relativas aos comandos de "manipulação da heap", são também modificadas de forma a permitir a admissibilidade da regra da consequência. Importa ainda salientar que, com a eliminação da regra da consequência, chegamos a um sistema onde é possível mecanizar a tarefa de geração de um conjunto de condições necessárias e suficientes, que reflectem a validade de um triplo de *Hoare*. Ao longo deste documento, quando necessitarmos de uma v variável fresca, isto é, que não ocorre nem no comando nem na pré-condição nem na pós-condição, assinalamos isso com $v\ fresh$.

$$\frac{\models \phi \Rightarrow \psi}{\{\phi\}\texttt{skip}\{\psi\}}\ \text{Skip}$$

$$\frac{\models \phi \Rightarrow \psi[a/v]}{\{\phi\}v{:=}a\{\psi\}}\ \text{Atrib.}$$

$$\frac{\{\phi\}\mathsf{C}_1\{\theta\} \qquad \{\theta\}\mathsf{C}_2\{\psi\}}{\{\phi\}\mathsf{C}_1;\mathsf{C}_2\{\psi\}}\ \text{Comp.}$$

$$\frac{\{\phi \wedge b\}\mathsf{C}_t\{\psi\} \qquad \{\phi \wedge \neg b\}\mathsf{C}_f\{\psi\}}{\{\phi\}\texttt{if}\ b\ \texttt{then}\ \mathsf{C}_t\ \texttt{else}\ \mathsf{C}_f\{\psi\}}\ \text{If}$$

$$\frac{\models \phi \Rightarrow \theta \qquad \{\theta \wedge b\}\mathsf{C}\{\theta\} \qquad \models \theta \wedge \neg b \Rightarrow \psi}{\{\phi\}\texttt{while}\ b\ \{\mathsf{C}\}\{\psi\}}\ \text{While}$$

$$\frac{\models \phi \Rightarrow (p \mapsto [-] * (p \mapsto [f:a] \mathrel{-\!\!*} \psi))}{\{\phi\}p{\rightarrow}f{:=}a\{\psi\}}\ \text{Mutation}$$

$$\frac{\models \phi \Rightarrow (p \mapsto [-] * \psi)}{\{\phi\}\texttt{dispose}(p)\{\psi\}}\ \text{Dispose}$$

$$\frac{p' fresh \qquad \models \phi \Rightarrow (\forall p'.p' \mapsto [-] \mathrel{-\!\!*} \psi[p'/p])}{\{\phi\}\texttt{new}(p)\{\psi\}}\ \text{Allocation}$$

$$\frac{v_1 fresh \qquad \models \phi \Rightarrow \exists v_1.(\psi[v_1/v] \wedge p \hookrightarrow [f:v_1])}{\{\phi\}v{:=}p \rightarrow f\{\psi\}}\ \text{Lookup}$$

Figura 4.2: $\mathbf{S^{gb}}$- Sistema *goal-directed*

A regra da consequência necessita de espalhar as suas condições laterais por todos os restantes casos. Deste modo a regra será admissível, e este sistema notado por $\mathbf{S^{gb}}$ passa a ser *goal directed*. As regras de $\mathbf{S^{gb}}$ estão na Figura 4.2.

Serão agora enunciados e provados os teoremas que demonstram a equivalência entre o Sistema $\mathbf{S^g}$ e o sistema $\mathbf{S^{gb}}$.

Teorema 4.2.1 Se $\vdash_{S^{gb}} \{\phi\}\mathsf{C}\{\psi\}$, então $\vdash_{S^g} \{\phi\}\mathsf{C}\{\psi\}$.

Demonstração Por indução nas derivações de $\vdash_{S^{gb}} \{\phi\}\mathsf{C}\{\psi\}$:

1. Caso *mutation*:

$$\frac{\models \phi \Rightarrow p \mapsto [-] * (p \mapsto [f:a] \mathrel{-\!\!*} \psi)}{\{\phi\}p{\rightarrow}f{:=}a\{\psi\}}\ \text{Mutation}$$

   Queremos demonstrar que $\vdash_{S^g} \{\phi\}p{\rightarrow}f{:=}a\{\psi\}$.

   Ora, basta então tomar,

$$\dfrac{\dfrac{}{\{p \mapsto [f:-] * (p \mapsto [f:a] \mathrel{-\!\!*} \psi)\}p{\rightarrow}f{:=}a\{p \mapsto [f:a] * (p \mapsto [f:a] \mathrel{-\!\!*} \psi)\}}\ \text{Mutation}}{\{\phi\}p{\rightarrow}f{:=}a\{\psi\}}\ \text{Conseq.}$$

   Ao nível das condições laterais da regra da Conseq., temos que garantir, $\models \phi \Rightarrow p \mapsto [f:-] * (p \mapsto [f:a] \mathrel{-\!\!*} \psi)$ e $\models p \mapsto [f:a] * (p \mapsto [f:a] \mathrel{-\!\!*} \psi) \Rightarrow \psi$. A primeira segue de imediato da hipótese de indução e de $p \mapsto [-] \Leftrightarrow p \mapsto [f:-]$. A segunda é consequência do Lema A.1.1 (onde $\phi = p \mapsto [f:a]$).

2. Caso *allocation*:

$$\frac{\models \phi \Rightarrow (\forall p'.p' \mapsto [-] \mathrel{-\!\!*} \psi[p'/p])}{\{\phi\}\mathtt{new}(p)\{\psi\}}\ \text{Allocation}$$

   Queremos demonstrar que $\vdash_{S^g} \{\phi\}\mathtt{new}(p)\{\psi\}$.

   Basta então tomar,

$$\dfrac{\dfrac{\dfrac{\dfrac{p \notin free(\forall p'.p' \mapsto [-] \mathrel{-\!\!*} \psi[p'/p])}{\{\forall p'.p \mapsto [-] \mathrel{-\!\!*} \psi[p'/p]\}\mathtt{new}(p)\{p \mapsto [-] * (\forall p'.p' \mapsto [-] \mathrel{-\!\!*} \psi[p'/p])\}}\ \text{Allocation}}{\{\forall p'.p \mapsto [-] \mathrel{-\!\!*} \psi[p'/p]\}\mathtt{new}(p)\{p \mapsto [-] * (p \mapsto [-] \mathrel{-\!\!*} \psi)\}}\ \text{Conseq.3}}{\{\forall p'.p \mapsto [-] \mathrel{-\!\!*} \psi[p'/p]\}\mathtt{new}(p)\{\psi\}}\ \text{Conseq.2}}{\{\phi\}\mathtt{new}(p)\{\psi\}}\ \text{Conseq.1}$$

   (a) Conseq 1- A condição $\models \phi \Rightarrow \forall p'.p \mapsto [-] \mathrel{-\!\!*} \psi[p'/p]$ foi suposta inicialmente como verdadeira.

   (b) Conseq 2- A condição $\models p \mapsto [-] * (p \mapsto [-] \mathrel{-\!\!*} \psi) \Rightarrow \psi$ segue do Lema A.1.1.

(c) Conseq 3- A condição $\models p \mapsto [-] * (\forall p'.p' \mapsto [-] \mathrel{-\!\!*} \psi[p'/p]) \Rightarrow p \mapsto [-] * (p \mapsto [-] \mathrel{-\!\!*} \psi)$ segue do Lema A.1.13

(d) Allocation - A condição lateral necessária a aplicação da regra *allocation*, isto é, $p \notin free(\forall p'.p' \mapsto [-] \mathrel{-\!\!*} \psi[p'/p])$. Isto é óbvio visto que p' é uma variável fresca.

3. Caso *dispose*

$$\frac{\models \phi \Rightarrow (p \mapsto [-] * \psi)}{\{\phi\}\texttt{dispose}(p)\{\psi\}}\ \text{Dispose}$$

Queremos demonstrar que $\vdash_{S^g} \{\phi\}\texttt{dispose}(p)\{\psi\}$.

Ora, basta então tomar,

$$\dfrac{\dfrac{}{\{p \mapsto [-] * \psi\}\texttt{dispose}(p)\{\psi\}}\ \text{Dispose}}{\{\phi\}\texttt{dispose}(p)\{\psi\}}\ \text{Conseq.}$$

Falta apenas demonstrar a condição lateral necessária à aplicação da regra da consequência, ou seja, $\models \phi \Rightarrow (p \mapsto [-] * \psi)$, mas isto é uma suposição inicial.

4. Caso *lookup*

$$\frac{\models \phi \Rightarrow \exists v_1.(\psi[v_1/v] \wedge p \hookrightarrow [f : v_1])}{\{\phi\}v{:=}p \rightarrow f\{\psi\}}\ \text{Lookup}$$

Queremos demonstrar que $\vdash_{S^g} \{\phi\}v{:=}p \rightarrow f\{\psi\}$ com $v_1 fresh$. Suponhamos que $\models \phi \Rightarrow \exists v_1.(\psi[v_1/v] \wedge p \hookrightarrow [f : v_1])$. Tomando $v_2 fresh$, podemos construir a derivação:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\{\exists v_1.((p \mapsto [f : v_1]) * ((p[v_2/v] \mapsto [f : v_1] \mathrel{-\!\!*} \psi[v_1/v])[v/v_2]))\}v{:=}p \rightarrow f\{\exists v_2.((p[v_2/v] \mapsto [f : v_1]) * ((p[v_2/v] \mapsto [f : v_1] \mathrel{-\!\!*} \psi[v_1/v])[v/v_1]))\}}\ \text{Lookup}}{\{\exists v_1.((p \mapsto [f : v_1]) * ((p[v_2/v] \mapsto [f : v_1] \mathrel{-\!\!*} \psi[v_1/v])[v/v_2]))\}v{:=}p \rightarrow f\{\exists v_2.((p[v_2/v] \mapsto [f : v_1]) * ((p[v_2/v] \mapsto [f : v_1] \mathrel{-\!\!*} \psi[v_1/v])[v/v_1]))\}}\ \text{Conseq. 5}}{\{\exists v_1.((p \mapsto [f : v_1]) * (p \mapsto [f : v_1] \mathrel{-\!\!*} \psi[v_1/v]))\}v{:=}p \rightarrow f\{\exists v_2.((p[v_2/v] \mapsto [f : v_1]) * (p[v_2/v] \mapsto [f : v_1] \mathrel{-\!\!*} \psi))\}}\ \text{Conseq. 4}}{\{\exists v_1.((p \mapsto [f : v_1]) * (p \mapsto [f : v_1] \mathrel{-\!\!*} \psi[v_1/v]))\}v{:=}p \rightarrow f\{\exists v_2.(\psi)\}}\ \text{Conseq. 3}}{\{\exists v_1.((p \mapsto [f : v_1]) * (p \mapsto [f : v_1] \mathrel{-\!\!*} \psi[v_1/v]))\}v{:=}p \rightarrow f\{\psi\}}\ \text{Conseq. 2}}{\{\exists v_1.(\psi[v_1/v] \wedge p \hookrightarrow [f : v_1])\}v{:=}p \rightarrow f\{\psi\}}\ \text{Conseq. 1}}{\{\phi\}v{:=}p \rightarrow f\{\psi\}}$$

Note-se que:

(a) Conseq. 1 - As condições laterais são $\models \phi \Rightarrow \exists v_1.(\psi[v_1/v] \wedge p \hookrightarrow [f : v_1])$ e $\models \psi \Rightarrow \psi$. A primeira foi suposta inicialmente como verdadeira. A segunda é trivialmente verdadeira.

(b) Conseq. 2 - Na segunda aplicação da regra da consequência temos que garantir que $\models \exists v_1.(\psi[v_1/v] \wedge p \hookrightarrow [f : v_1]) \Rightarrow \exists v_1.(p \mapsto [f : v_1] * (p \mapsto [f : v_1] \mathrel{-\!\!*} \psi[v_1/v]))$ e ainda que $\models \psi \Rightarrow \psi$. A demonstração da primeira condição encontram-se em apêndice, no Lema A.1.7. A segunda condição é trivialmente verdadeira.

(c) Conseq. 3 - Trata-se de uma quantificação com recurso a uma variável fresca e portanto é verdadeira.

(d) Conseq. 4 - As condições laterais da regra da consequência são $\models \exists v_1.((p \mapsto [f : v_1]) * (p \mapsto [f : v_1] -\!\!* \psi[v_1/v])) \Rightarrow \exists v_1.((p \mapsto [f : v_1]) * (p \mapsto [f : v_1] -\!\!* \psi[v_1/v]))$ e ainda que $\models \exists v_2.((p[v_2/v] \mapsto [f : v_1]) * (p[v_2/v] \mapsto [f : v_1] -\!\!* \psi)) \Rightarrow \exists v_2.\psi$. A primeira é trivialmente verdadeira. A segunda encontra-se em apêndice no lema A.1.8 tomando $\phi = p[m/v] \mapsto [f : a]$ e $\psi = \psi$.

(e) Conseq. 5 - Na verdade, os triplos na premissa e na conclusão desta interferência são os mesmos.

(f) Lookup Falta agora provar que $v \notin free(p[v_2/v] \mapsto [f : v_1] -\!\!* \psi[v_1/v])$ mas é evidente pois todas as possíveis ocorrências de v são substituídas pois v_1 e v_2 são variáveis frescas. Teremos por fim que demonstrar que $v_1 \neq v \wedge v_2 \neq v \wedge v_1 \neq p$, mas de sabermos que v_1, v_2 são variáveis frescas temos em particular aquilo que pretendíamos demonstrar.

Os restantes casos da prova são análogos aos descritos em [24] (na relação entre o sistema $\mathscr{H}_g$ e o sistema $\mathscr{H}$).

□

Teorema 4.2.2 Se $\vdash_{Sg} \{\phi\}\mathsf{C}\{\psi\}$, então $\vdash_{Sgb} \{\phi\}\mathsf{C}\{\psi\}$.

Demonstração Por indução nas derivações de $\vdash_{Sg} \{\phi\}\mathsf{C}\{\psi\}$.

1. Caso *mutation*:

$$\frac{}{\{p \mapsto [f : -] * \theta\}p{\rightarrow}f{:=}a\{p \mapsto [f : a] * \theta\}}\ \text{Mutation}$$

Queremos demonstrar que $\vdash_{Sgb} \{p \mapsto [f : -] * \theta\}p{\rightarrow}f{:=}a\{p \mapsto [f : a] * \theta\}$.

Ora, basta então tomar,

$$\frac{\models (p \mapsto [-] * \theta) \Rightarrow (p \mapsto [-] * (p \mapsto [f : a] -\!\!* (p \mapsto [f : a] * \theta))}{\{p \mapsto [f : -] * \theta\}p{\rightarrow}f{:=}a\{p \mapsto [f : a] * \theta\}}\ \text{Mutation}$$

Falta agora demonstrar que $\models (p \mapsto [-] * \theta) \Rightarrow (p \mapsto [-] * (p \mapsto [f : a] -\!\!* (p \mapsto [f : a] * \theta))$. Segue do Lema A.1.2.

2. Caso *dispose*:

$$\frac{}{\{p \mapsto [-] * \theta\}\texttt{dispose}(p)\{\theta\}}\ \text{Dispose}$$

Queremos demonstrar que $\vdash_{S^{gb}} \{p \mapsto [-] * \theta\}\texttt{dispose}(p)\{\theta\}$.

Basta então tomar,

$$\frac{\models (p \mapsto [-] * \theta) \Rightarrow (\theta * p \mapsto [-])}{\{p \mapsto [-] * \theta\}\texttt{dispose}(p)\{\theta\}}\ \text{Dispose}$$

Falta agora demonstrar que $\models (p \mapsto [-] * \theta) \Rightarrow \theta * p \mapsto [-])$, que é verdadeira uma vez que a conjunção separada é comutativa.

3. Caso *allocation*:

$$\frac{p \notin free(\theta)}{\{\theta\}\texttt{new}(p)\{p \mapsto [-] * \theta\}}\ \text{Allocation}$$

Queremos demonstrar que $\vdash_{S^{gb}} \{\theta\}\texttt{new}(p)\{p \mapsto [-] * \theta\}$.
Basta então tomar,

$$\frac{\models \theta \Rightarrow (\forall p'.p' \mapsto [-] -\!\!* (p \mapsto [-] * \theta)[p'/p])}{\{\theta\}\texttt{new}(p)\{p \mapsto [-] * \theta\}}\ \text{Allocation}$$

Falta agora demonstrar que $\models \theta \Rightarrow (\forall p'.p' \mapsto [-] -\!\!* (p \mapsto [-] * \theta)[p'/p])$. Suponhamos $[\![\theta]\!](s,h) = T$.

Queremos demonstrar que $[\![(\forall p'.p' \mapsto [-] -\!\!* (p \mapsto [-] * \theta)[p'/p])]\!](s,h) \Leftrightarrow [\![p' \mapsto [-] -\!\!* (p \mapsto [-] * \theta)[p'/p])]\!]([s|p' : l],h)$ para todo o l.
Suponhamos que $h_1 \# h$ e $[\![p' \mapsto [-]]\!]([s|p' : l],h_1)$.
Queremos demonstrar que $[\![(p \mapsto [-] * \theta)[p'/p]]\!]([s|p' : l],h_1.h)$.
De sabermos que $p \notin free(\theta)$, temos então que mostrar $[\![p' \mapsto [-] * \theta]\!]([s|p' : l],h_1.h)$ para o que basta mostrar $[\![p' \mapsto [-]]\!]([s|p' : l],h_1) \wedge [\![\theta]\!]([s|p' : l],h)$.
A primeira condição foi suposta inicialmente como verdadeira. A segunda segue de sabermos p' fresh, o que significa que p' não ocorre em θ e portanto $[\![\theta]\!]([s|p' : l],h)$, sendo que está última foi suposta inicialmente como verdadeira.

4. Caso *lookup*

$$\frac{v \notin free(\theta) \wedge v_1 \neq v_2 \wedge v_1 \neq v \wedge v_2 \neq v \wedge v_1 \neq p \wedge v_2 \neq p}{\{\exists v_1.(p \mapsto [f : v_1] * (\theta[v/v_2]))\}v{:=}p \rightarrow f\{\exists v_2.((p[v_2/v] \mapsto [f : v]) * (\theta[v/v_1]))\}}\ \text{Lookup}$$

Queremos demonstrar que $\vdash_{S^{gb}} \{\exists v_1.(p \mapsto [f : v_1] * (\theta[v/v_2]))\}v{:=}p \rightarrow f\{\exists v_2.((p[v_2/v] \mapsto [f : v]) * (\theta[v/v_1]))\}$.
Basta então tomar,

$$D' = \frac{}{\{\exists v_1.(p \mapsto [f : v_1] * (\theta[v/v_2]))\}v{:=}p \rightarrow f\{\exists v_2.((p[v_2/v] \mapsto [f : v]) * (\theta[v/v_1]))\}}\ \text{Lookup}$$

Falta agora demonstrar que $\models \exists v_1.p \mapsto [f:v_1] * (\theta[v/v_2]) \Rightarrow \exists v_3.(\exists v_2.(p[v_2/v] \mapsto [f:v]) * (\theta[v/v_1])[v_3/v] \wedge p \hookrightarrow [f:v_3])$

Suponhamos que $[\![\exists v_1.p \mapsto [f:v_1] * \theta[v/v_2]]\!](s,h)$, ou seja,
i)$dom(h_1) = \{[\![p]\!]([s|v_1:a])\}$∧ii) $(h_1([\![p]\!]([s|v_1:a]))(f) = a$∧ iii) $[\![\theta]\!]([s|v_1 : a|v_2 : [\![v]\!](s,,)h_2)$.

Queremos demonstrar que $[\![\exists v_3.(\exists v_2.(p[v_2/v] \mapsto [f:v]) * (\theta[v/v_1])[v_3/v] \wedge p \hookrightarrow [f:v_3])]\!](s,h)$, isto é, $\Leftrightarrow [\![\exists v_2.(p[v_2/v] \mapsto [f:v]) * (\theta[v/v_1])[v_3/v] \wedge p \hookrightarrow [f : v_3])]\!]([s|v_3:a',h)$ para algum a'
$\Leftrightarrow \exists h_1' \# h_2' \wedge h = h_1'.h_2 \wedge [\![p[v_2/v] \mapsto [f:v]]\!]([s|v_3:a'|v_2:a''], h_1') \wedge [\![(\theta[v/v_1])[v_3/v]]\!]([s|v_3 : a'|v_2 : a''], h_2) \wedge [\![p \hookrightarrow [f:v_3]]\!]([s|v_3:a'], h)$ para algum a', a''.
Tomando $a' = a$, $a'' = [\![v]\!](s)$,$h_1' = h_1$ e $h_2' = h_2$, temos que

(a) $dom(h_1) = \{[\![p[v_2/v]]\!]([s|v_3:a|v_2:[\![v]\!](s)])\} \Leftrightarrow dom(h_1) = \{[\![p]\!]([s|v_3 : a|v_2 : [\![v]\!](s)|v : [\![v]\!](s)])\}$ e como $p \neq v_2, p \neq v_1$ e $v_3 fresh$ então por i) temos que é verdadeiro por suposição.

(b) $(h_1([\![p[v_2/v]]\!]([s|v_3:a|v_2:[\![v]\!](s,)])))(f) = [\![v_3]\!]([s|v_3:a|v_2:[\![v]\!](s)]) \Leftrightarrow (h_1([\![p]\!]([s|v_3:a|v_2:[\![v]\!](s)|v:[\![v_2]\!]([s|v_3:a|v_2:[\![v]\!](s)])])))(f) = [\![v_3]\!]([s|v_3 : a])$.

De $p \neq v_1, p \neq v_2$, $v_3 fresh$ e de ii) temos que a afirmação é verdadeira.

(c) $[\![\theta[v_3/v_1]]\!]([s|v_3:a|v_2:[\![v]\!](s)], h_2) \Leftrightarrow [\![\theta]\!]([s|v_3:a|v_2:[\![v]\!](s)|v_1:a], h_2)$ como $v_3 fresh$ logo $v_3 \notin free(\theta)$ e por iii) temos que é verdadeira.

(d) $[\![p \hookrightarrow [f:v_3]]\!]([s|v_3:a'], h) \Leftrightarrow \exists h_1' \# h_2 \; h = h_1.h_2 \; [\![p \mapsto [f:v_3]]\!]([s|v_3 : a'], h_1) \wedge [\![T]\!]([s|v_3:a'], h_2)$. Tomando $h_1' = h_1$, $h_2' = h_2$, a segunda propriedade segue de imediato pela definição. $[\![p \mapsto [f:v_3]]\!]([s|v_3:a'], h_1) \Leftrightarrow dom(h_1) = \{[\![p]\!]([s|v_3:a'])\} \wedge (h_1([\![p]\!]([s|v_3:a'])))(f) = [\![v_3]\!]([s|v_3 : a'])$. De $p \neq v_3, p \neq v_1$ e de i) e ii) temos o pretendido.

5. Caso consequência: Este caso segue da admissibilidade da regra da consequência no sistema $\mathbf{S^{gb}}$, o que se encontra provado no Proposição 4.2.3 abaixo.

Os casos relativos às regras da linguagem *While* simples, encontra-se em [24] (na relação entre o sistema formal $\mathscr{H}$ e o sistema formal $\mathscr{H}_g$).

□

O resultado seguinte estabelece a admissibilidade da regra da consequência no sistema $\mathbf{S^{gb}}$. Note-se que em $\mathbf{S^g}$ já não temos a regra de *frame* e da eliminação de variáveis auxiliares e, por isso, não temos o problema de "distribuir" a regra da consequência por estas regras.

Proposição 4.2.3 Se $\vdash_{S^{gb}} \{\phi\}\mathsf{C}\{\psi\}$, $\models \phi' \Rightarrow \phi$ e $\models \psi \Rightarrow \psi'$, então $\vdash_{S^{gb}} \{\phi'\}\mathsf{C}\{\psi'\}$.

Demonstração Por indução nas derivações de $\vdash_{S^{gb}} \{\phi\}\mathsf{C}\{\psi\}$:

1. Caso *mutation*:

$$\frac{\models \phi \Rightarrow (p \mapsto [-] * (p \mapsto [f:a] \mathrel{-\!\!*} \psi))}{\{\phi\} p{\rightarrow}f{:=}a \{\psi\}} \text{ Mutation}$$

Queremos demonstrar que se $\models \phi' \Rightarrow \phi$ e $\models \psi \Rightarrow \psi'$, então $\vdash_{S^{gb}} \{\phi'\} p{\rightarrow}f{:=}a \{\psi'\}$.

Ora, basta então tomar,

$$\frac{\models \phi' \Rightarrow (p \mapsto [-] * (p \mapsto [f:a] \mathrel{-\!\!*} \psi'))}{\{\phi'\} p{\rightarrow}f{:=}a \{\psi'\}} \text{ Mutation}$$

Falta agora demonstrar que $\models \phi' \Rightarrow (\psi' * \mathrm{p} \mapsto [-])$, mas de sabermos que $\models \phi \Rightarrow \phi$ e de $\models \phi \Rightarrow (p \mapsto [-] * (p \mapsto [f:a] \mathrel{-\!\!*} \psi))$ temos que $\models \phi' \Rightarrow (p \mapsto [-] * (p \mapsto [f:a] \mathrel{-\!\!*} \psi'))$. Daqui, em apêndice no Lema A.1.9.

2. Caso *dispose*:

$$\frac{\models \phi \Rightarrow (p \mapsto [-] * \psi)}{\{\phi\} \texttt{dispose}(p) \{\psi\}} \text{ Dispose}$$

Queremos demonstrar que se $\models \phi' \Rightarrow \phi$ e $\models \psi \Rightarrow \psi'$, então $\vdash_{S^{gb}} \{\phi'\} \texttt{dispose}(p) \{\psi'\}$.

Ora, basta então tomar,

$$\frac{\models \phi' \Rightarrow (\psi' * p \mapsto [-])}{\{\phi'\} \texttt{dispose}(p) \{\psi'\}} \text{ Dispose}$$

Falta agora demonstrar que $\models \phi' \Rightarrow (\psi' * p \mapsto [-])$.

De $\models \phi \Rightarrow (\psi * p \mapsto [-])$ e $\models \phi' \Rightarrow \phi$ pela transitividade da relação de implicação, temos que $\models \phi' \Rightarrow (\psi * p \mapsto [-])$. Daqui, pelo Lema A.1.10, segue a condição pedida.

3. Caso *Allocation*:

$$\frac{\models \phi \Rightarrow (\forall p'.p' \mapsto [-] \mathrel{-\!\!*} \psi[p'/p])}{\{\phi\} \texttt{new}(p) \{\psi\}} \text{ Allocation}$$

Queremos demonstrar que se $\models \phi' \Rightarrow \phi$ e $\models \psi \Rightarrow \psi'$, então $\vdash_{S^{gb}} \{\phi'\} \texttt{new}(p) \{\psi'\}$. Basta então tomar,

$$\frac{\models \phi' \Rightarrow (\forall \mathrm{x}'.\mathrm{x}' \mapsto [-] * \psi'[\mathrm{x}'/p])}{\{\phi'\} \texttt{new}(p) \{\psi'\}} \text{ Allocation}$$

Falta agora demonstrar que $\models \phi' \Rightarrow (\forall p'.p' \mapsto [-] * \psi'[p'/p])$.

De sabermos que $\models \phi \Rightarrow (\forall p'.p' \mapsto [-] * \psi[p'/p])$ e $\models \phi' \Rightarrow \phi$, pela transitividade da relação de implicação, temos que $\models \phi' \Rightarrow (\forall p'.p' \mapsto [-] * \psi[p'/p])$.

Daqui, pelo Lema A.1.11 em apêndice segue a condição pedida.

4. Caso *lookup*

$$\frac{\models \phi \Rightarrow \exists v_1.(\psi[v_1/v] \wedge p \hookrightarrow [f : v_1])}{\{\phi\} v{:=}p \rightarrow f\{\psi\}} \text{ Lookup}$$

Queremos demonstrar que se $\models \phi' \Rightarrow \phi$ e $\models \psi \Rightarrow \psi'$, então $\vdash_{S^{gb}} \{\phi'\} v{:=}p \rightarrow f\{\psi'\}$.

Basta então tomar,

$$\frac{\models \phi' \Rightarrow \exists v_1.(\psi'[v_1/v] \wedge p \hookrightarrow [f : v_1])}{\{\phi'\} v{:=}p \rightarrow f\{\psi'\}} \text{ Lookup}$$

Falta agora demonstrar que $\models \phi' \Rightarrow \exists v_1.(\psi'[v_1/v] \wedge p \hookrightarrow [f : v_1])$.

De sabermos $\models \phi \Rightarrow \exists v_1.(\psi[v_1/v] \wedge p \hookrightarrow [f : v_1])$ e $\models \phi' \Rightarrow \phi$, pela transitividade da relação de implicação, temos que $\models \phi' \Rightarrow \exists v_1.(\psi[v_1/v] \wedge p \hookrightarrow [f : v_1])$. Daqui, pelo Lema A.1.12, em apêndice segue o pretendido.

Os restantes casos desta prova, são relativos aos comandos da linguagem *While* e as provas são essencialmente iguais às descritas em [24] (na admissibilidade da regra da consequência no sistema $\mathscr{H}_g$).

□

4.3 VCGen - gerador de condições de verificação

Tendo por base o sistema *goal directed*, $\mathbf{S^{gb}}$, vamos agora apresentar um *VCGen* que dado um triplo de Hoare com os ciclos anotados com os invariantes, gera as condições de verificação que, sendo válidas, garantem que existe uma derivação para o triplo que usa aqueles invariantes.

O sistema $\mathbf{S^{gb}}$ tem em si implícita uma estratégia para construir provas de um modo determinístico. No caso da regra de composição (isto é, da sequenciação de comandos), quando esta regra é aplicada a um dado triplo, são gerados dois objectivos de prova (correspondentes às duas premissas das regras): o primeiro com uma pós-condição desconhecida e o segundo com uma pré-condição desconhecida.

O método de prova em que vamos basear o nosso *VCGen* consiste em focarmos-nos em primeiro lugar no segundo objectivo de prova, descobrindo a pré-condição

inicialmente desconhecida e que depois vai ser propagada para o primeiro objectivo de prova, uma vez que será a pós-condição desse triplo.

O *VCGen* que vamos apresentar baseia-se na *weakest precondition*, que determina a pré-condição mais fraca para assegurar a validade da pós-condição após a execução de um comando [3]. Há ainda um outro aspecto que introduz uma dificuldade adicional na construção de uma derivação para um triplo de Hoare que se prende com os ciclos While. No processo de construção de prova é necessário identificar os invariantes de ciclo apropriados. Esses invariantes não são gerados automaticamente a partir da pré e pós-condição do programa, e necessitam de ser descobertos.

É pratica corrente na verificação de programas que os ciclos While sejam anotados com invariantes de ciclo, e no processo de prova de correcção desse programa face a uma pré e pós-condição, serão esses invariantes a ser usados nas provas relativas aos ciclos.

Portanto, enquanto na apresentação *standard* (que fizemos no Capítulo 3) um triplo de Hoare pode ser derivado desde que existam invariantes, adequados à prova de correcção desse triplo, na prática (quando escolhemos os invariantes) a derivação só vai ser possível se os ciclos forem anotados com invariantes adequados a essa prova.

Como estamos interessados na automatização do processo de prova e os invariantes dos ciclos não podem ser inferidos automaticamente, vamos considerar uma $While_{pr}$ com os ciclos anotados onde o comando While passa a ter a forma $\texttt{while}\ b\ [\theta]\{\mathsf{C}\}$, sendo o θ o invariante que anota o ciclo. Note-se que a anotação não altera em nada a semântica. Esta abordagem também pode ser encontrada em [3].

Portanto, a regra para o ciclo While do sistema $\mathbf{S^{gb}}$ escreve-se agora,

$$\frac{\models \phi \Rightarrow \theta \qquad \{\theta \wedge b\}\mathsf{C}\{\theta\} \qquad \models \theta \wedge \neg b \Rightarrow \psi}{\{\phi\}\texttt{while}\ b\ [\theta]\{\mathsf{C}\}\{\psi\}}$$

Em resumo, o algoritmo *VCGen* para um dado triplo $\{\phi\}\mathsf{C}\{\psi\}$, (em que C é um programa de ciclos anotados) vai primeiro calcular a pré-condição mais fraca de C relativamente a ψ, $wp(\mathsf{C},\psi)$, e depois gerar ainda a condição $\phi \Rightarrow wp(\mathsf{C},\psi)$ para garantir que o triplo é correcto. Este algoritmo é baseado no apresentado em [3]. A *weakest- precondition* de um programa C relativamente à pós-condição ψ, $wp(\mathsf{C},\psi)$, está definida na Figura 4.3. Note-se que como estamos a trabalhar com um ciclo anotado, a *weakest- precondition* do ciclo é o próprio invariante.

$wp(\texttt{skip}, \psi) = \psi$
$wp(v{:=}a, \psi) = \psi[a/v]$
$wp(\mathsf{C}_1;\mathsf{C}_2, \psi) = wp(\mathsf{C}_1, wp(\mathsf{C}_2, \psi))$
$wp(\texttt{if } b \texttt{ then } \mathsf{C}_t \texttt{ else } \mathsf{C}_f, \psi) = (b \to wp(\mathsf{C}_t, \psi)) \wedge (\neg b \to wp(\mathsf{C}_f, \psi))$
$wp(\texttt{while } b\ [\theta]\{\mathsf{C}\}, \psi) = \theta$
$wp(\texttt{new}(p), \psi) = \forall p'.(p' \mapsto [-] \mathrel{-\!\!*} \psi[p'/p])$ (com $p' fresh$)
$wp(p{\to}f{:=}a, \psi) = (p \mapsto [-] * (p \mapsto [f : a] \mathrel{-\!\!*} \psi))$
$wp(v{:=}p \to f, \psi) = \exists v_1.(\psi[v_1/v] \wedge p \hookrightarrow [f : v_1])$ (com $v_1 fresh$)
$wp(\texttt{dispose}(p), \psi) = \psi * (p \mapsto [-])$

Figura 4.3: *wp- weakest pre-condition*

Exemplo 4.3.1 Retomemos o exemplo do início do capítulo de forma a ilustrar a aplicação das regras da estratégia *weakest pre-condition*

$wp(\texttt{new}(p); p \to valor := x; x := y; y := p \to valor; \texttt{dispose}(p), x = 2 \wedge y = 1)$
$\ldots$
$\Leftrightarrow \forall x'.x' \mapsto [-] \mathrel{-\!\!*} (x' \mapsto [-] * (x' \mapsto [valor : x] \mathrel{-\!\!*} (\exists v_1.y = 2 \wedge v_1 = 1 \wedge x' \hookrightarrow [valor : v_1])))$
$\ldots$
$\Leftrightarrow x = 1 \wedge y = 2$

Este cálculo responde à pergunta exposta no início deste capítulo: realmente, a condição apresentada $x = 1 \wedge y = 2$ é a condição mais fraca para garantir $x = 2 \wedge y = 1$ após a execução.

É necessário agora garantir que $wp(\mathsf{C}, \psi)$ é mesmo a pré-condição mais fraca de C face a ψ. Baseados no resultado apresentado em [3], vamos agora estabelecer o mesmo resultado para a lógica da separação.

Teorema 4.3.2 Se $\vdash_{S^{gb}} \{\phi\}\mathsf{C}\{\psi\}$ então

1. $\vdash_{S^{gb}} \{wp(\mathsf{C}, \psi)\}\mathsf{C}\{\psi\}$
2. $\models \phi \Rightarrow wp(\mathsf{C}, \psi)$

Demonstração Por indução na dedução de $\vdash_{S^{gb}} \{\phi\}\mathsf{C}\{\psi\}$:

1. Caso *mutation*:

$$D = \frac{\models \phi \Rightarrow (p \mapsto [-] * (p \mapsto [f : a] \mathrel{-\!\!*} \psi))}{\{\phi\}p{\to}f{:=}a\{\psi\}}\ \text{Mutation}$$

Queremos demonstrar que

(a) $\vdash_{S^{gb}} \{wp(p{\rightarrow}f{:=}a, \psi)\}p{\rightarrow}f{:=}a\{\psi\}$

(b) $\models \phi \Rightarrow wp(p{\rightarrow}f{:=}a, \psi)$

Para mostrar (a) construímos a derivação,

$$\frac{(\mathrm{p} \mapsto [-] * (\mathrm{p} \mapsto [\mathrm{f} : \mathrm{a}] \mathrel{-\!\!*} \psi)) \Rightarrow (\mathrm{p} \mapsto [-] * (\mathrm{p} \mapsto [\mathrm{f} : \mathrm{a}] \mathrel{-\!\!*} \psi))}{\{wp(p{\rightarrow}f{:=}a, \psi)\}p{\rightarrow}f{:=}a\{\psi\}} \text{ Mutation}$$

Note-se que a condição lateral, tem a estutura $\theta \Rightarrow \theta$ e portanto é válida.

(b) Ora,

$$\begin{array}{l} \models \phi \Rightarrow wp(p{\rightarrow}f{:=}a, \psi) \\ \Leftrightarrow \models \phi \Rightarrow (p \mapsto [-] * (p \mapsto [f : a] \mathrel{-\!\!*} \psi)) \end{array}$$

e isto segue de termos assumido a existência da dedução D.

2. Caso *dispose*:

$$D = \frac{\models \phi \Rightarrow (p \mapsto [-] * \psi)}{\{\phi\}\texttt{dispose}(p)\{\psi\}} \text{ Dispose}$$

Queremos demonstrar que

(a) $\vdash_{S^{gb}} \{wp(\texttt{dispose}(p), \psi)\}\texttt{dispose}(p)\{\psi\}$

(b) $\models \phi \Rightarrow wp(\texttt{dispose}(p), \psi)$

Para (a), consideremos a dedução

$$\frac{\models (\psi * \mathrm{p} \mapsto [-]) \Rightarrow (\psi * \mathrm{p} \mapsto [-])}{\{wp(\texttt{dispose}(p), \psi)\}\texttt{dispose}(p)\{\psi\}} \text{ Dispose}$$

onde a condição lateral é uma fórmula válida.

Para mostrar (b), note-se que

$$\begin{array}{l} \models \phi \Rightarrow wp(\texttt{dispose}(p), \psi) \\ \Leftrightarrow \models \phi \Rightarrow (p \mapsto [-] * \psi) \end{array}$$

que segue da suposição relativa à dedução D.

3. Caso *allocation*:

$$D = \frac{\models \phi \Rightarrow (\forall p'.p' \mapsto [-] \mathrel{-\!\!*} \psi[p'/p])}{\{\phi\}\texttt{new}(p)\{\psi\}} \text{ Allocation}$$

Queremos demonstrar que

(a) $\vdash_{S^{gb}} \{wp(\mathtt{new}(p), \psi)\}\mathtt{new}(p)\{\psi\}$

(b) $\models \phi \Rightarrow wp(\mathtt{new}(p), \psi)$

Para mostrar (a), basta tomar,

$$D' = \frac{\models (\forall p'.p' \mapsto [-] \mathrel{-\!\!*} \psi[p'/p]) \Rightarrow (\forall p'.p' \mapsto [-] \mathrel{-\!\!*} \psi[p'/p])}{\{wp(\mathtt{new}(p), \psi)\}\mathtt{new}(p)\{\psi\}}\ \text{Allocation}$$

onde, de novo, a condição lateral é uma fórmula válida da forma $\theta \Rightarrow \theta$. Para mostrar (b), note-se que,

$$\models \phi \Rightarrow wp(\mathtt{new}(p), \psi)$$
$$\Leftrightarrow \models \phi \Rightarrow (\forall p'.p' \mapsto [-] \mathrel{-\!\!*} \psi[p'/p])$$

que segue da dedução D.

4. Caso *lookup*:

$$D = \frac{\models \phi \Rightarrow \exists v_1.(\psi[v_1/v] \wedge p \hookrightarrow [f : v_1])}{\{\phi\}v{:=}p \to f\{\psi\}}\ \text{Lookup}$$

Queremos demonstrar que

(a) $\vdash_{S^{gb}} \{wp(v{:=}p \to f, \psi)\}v{:=}p \to f\{\psi\}$

(b) $\models \phi \Rightarrow wp(v{:=}p \to f, \psi)$

Para mostrar (a), basta tomar,

$$\dfrac{\dfrac{\models \exists v_1.(\psi[v_1/v] \wedge p \hookrightarrow [f : v_1]) \Rightarrow \exists v_1.(\psi[v_1/v] \wedge p \hookrightarrow [f : v_1])}{\{\exists v_1.(\psi[v_1/v] \wedge p \hookrightarrow [f : v_1])\}v{:=}p \to f\{\psi\}}\ \text{Lookup}}{\{wp(v{:=}p \to f, \psi)\}v{:=}p \to f\{\psi\}}$$

Note-se que a condição lateral relativa à regra *lookup*, é trivialmente satisfeita.

Para mostrar (b),

$$\models \phi \Rightarrow wp(v{:=}p \to f, \psi)$$
$$\Leftrightarrow \models \phi \Rightarrow \exists v_1.(\psi[v_1/v] \wedge p \hookrightarrow [f : v_1])$$

que segue da existência da dedução D.

Os restantes casos seguem os casos apresentados na prova do resultado em [24].

Na Figura 4.4 define-se a função VC que, dado um programa e uma pós-condição, produz um conjunto de condições de verificação. Esta função segue de perto a estrutura das regras do sistema $\mathbf{S^{gb}}$, colectando as anotações laterais num conjunto. Note-se que no caso da sequenciação de comandos, a função VC segue a estratégia de *weakest-precondition* que descrevemos.
Note-se ainda que uma pré-condição ϕ numa condição lateral da forma $\phi \Rightarrow \psi$, para algum ψ, ϕ será tomada como sendo igual a ψ e portanto a condição é trivialmente satisfeita, pelo que estas condições não são incluídas no conjunto das condições geradas (pois não impõem qualquer condição adicional). Os algoritmos wp e VC estendem os descritos em [3], que foram descritos para tratar a linguagem *While* (sem memória dinâmica) acrescentando os casos relativos aos comandos que manipulam memória dinâmica.

$$
\begin{array}{l}
VC(\texttt{skip}, \psi) = \varnothing \\
VC(v{:=}a, \psi) = \varnothing \\
VC(\mathsf{C}_1;\mathsf{C}_2, \psi) = VC(\mathsf{C}_1, wp(\mathsf{C}_2, \psi)) \cup VC(\mathsf{C}_2, \psi) \\
VC(\texttt{if } b \texttt{ then } \mathsf{C}_t \texttt{ else } \mathsf{C}_f, \psi) = VC(\mathsf{C}_t, \psi) \cup VC(\mathsf{C}_f, \psi) \\
VC(\texttt{while } b\ [\theta]\{\mathsf{C}\}, \psi) = \{\theta \wedge b \rightarrow wp(\mathsf{C}, \theta), \theta \wedge \neg b \rightarrow \psi\} \cup VC(\mathsf{C}, \theta) \\
VC(\texttt{new}(p), \psi) = \varnothing \\
VC(p{\rightarrow}f{:=}a, \psi) = \varnothing \\
VC(v{:=}p \rightarrow f, \psi) = \varnothing \\
VC(\texttt{dispose}(p), \psi) = \varnothing
\end{array}
$$

Figura 4.4: VC

$$
VCGen(\{\phi\}\mathsf{C}\{\psi\}) = \{\phi \rightarrow wp(\mathsf{C}, \psi)\} \wedge VC(\mathsf{C}, \psi)
$$

Figura 4.5: $VCGen$- Algoritmo de geração das condições de verificação

Exemplo 4.3.3 Recorrendo ao cálculo executado no Exemplo 4.3.1, vejamos as condições de verificação geradas para o triplo apresentado como exemplo no inicio do capítulo temos:
$wp(\texttt{new}(p); p \rightarrow valor := x; x := y; y := p \rightarrow valor; \texttt{dispose}(p), x = 2 \wedge y = 1)$
(pelos cálculos efectuados em 4.3.1)
$\Leftrightarrow x = 1 \wedge y = 2$.

De $VC(\texttt{new}(p); p \rightarrow valor := x; x := y; y := p \rightarrow valor; \texttt{dispose}(p), x = 2 \wedge y = 1)$
$\Leftrightarrow \varnothing$.
Logo, $VCGen(\{x = 1 \wedge y = 2\}\texttt{new}(p); p \rightarrow valor := x; x := y; y := p \rightarrow valor; \texttt{dispose}(p); \{x = 2 \wedge y = 1\})$
$\Leftrightarrow x = 1 \wedge y = 2 \Rightarrow x = 1 \wedge y = 2$. Como é trivialmente verdadeira podemos então concluir que existe uma derivação para o triplo em causa.

□

Finalmente, apresenta-se na Figura 4.5 a função *VCGen* que calcula o conjunto de condições de verificação, com recurso às funções *VC* e *wp*. Como já foi explicado, o que esta função faz é acrescentar às condições geradas pelo *VC* uma condição extra $\phi \Rightarrow wp(\mathsf{C}, \psi)$ que garante que a pré-condição do triplo é suficiente para assegurar a pré-condição mais fraca de C face a ψ.

Resta agora demonstrar que este algoritmo de *VCGen* é adequado à sua tarefa, e que pode, portanto, ser usado para testar se um triplo é válido. Para isso temos que estabelecer uma equivalência entre a validade das condições geradas pelo *VCGen* e a dedutibilidade no sistema $\mathbf{S^{gb}}$ do triplo em causa. Isto é, iremos mostrar que todas as condições geradas serão válidas se e só se o triplo é derivável. Mostra-se em primeiro lugar que a validade do conjunto das formulas gerada pelo *VCGen* para um triplo implica a dedutibilidade desse triplo no sistema $\mathbf{S^{gb}}$. Em rigor é ainda necessário ter em atenção a questão das anotações dos comandos *while* (os invariantes de ciclo) no *VCGen*. Dado um comando C, C^- representará o comando obtido de C apagando as anotações dos comandos *while*.

Notação 4.3.4 Usaremos $\models VCGen(\{\phi\}\mathsf{C}\{\psi\})$ para indicar que todas as fórmulas do conjunto $VCGen(\{\phi\}\mathsf{C}\{\psi\})$ são válidas

Teorema 4.3.5 Se $\models VCGen(\{\phi\}\mathsf{C}\{\psi\})$, então $\vdash_{S^{gb}} \{\phi\}\mathsf{C}^-\{\psi\}$.

Demonstração Por indução em C.

1. Caso *mutation*:

   Suponhamos que $\models VCGen(\{\phi\}p{\rightarrow}f{:=}a\{\psi\})$, ou seja,

   $$\begin{array}{l} \models VCGen(\{\phi\}p{\rightarrow}f{:=}a\{\psi\}) \\ \Leftrightarrow \models \phi \Rightarrow (p \mapsto [-] * (p \mapsto [f:a] * \psi)) \end{array}$$

   Então, como pretendido, podemos construir a dedução:

   $$\frac{\models \phi \Rightarrow (p \mapsto [-] * (p \mapsto [f:a] \mathrel{-\!\!*} \psi))}{\{\phi\}p{\rightarrow}f{:=}a\{\psi\}}\ \text{Mutation}$$

   (Note-se que a eliminação de anotações em ciclos *while* não afecta o comando *mutation*).

2. Caso *dispose*:

   Suponhamos que $\models VCGen(\{\phi\}\texttt{dispose}(p)\{\psi\})$, ou seja,

   $$\begin{array}{l} \models VCGen(\{\phi\}\texttt{dispose}(p)\{\psi\}) \\ \Leftrightarrow \models \phi \Rightarrow \psi * p \mapsto [-] \end{array}$$

Assim,

$$\frac{\models \phi \Rightarrow (p \mapsto [-] * \psi)}{\{\phi\}\texttt{dispose}(p)\{\psi\}}\ \text{Dispose}$$

é uma dedução com o formato pretendido.

3. Caso *allocation*:

   Suponhamos que $\models VCGen(\{\phi\}\texttt{new}(p)\{\psi\})$, ou seja,

   $$\models VCGen(\{\phi\}\texttt{new}(p)\{\psi\})$$
   $$\Leftrightarrow \models \phi \Rightarrow (\forall p'.p' \mapsto [-] \mathrel{-\!\!*} \psi[p'/p])$$

   como tal podemos construir a dedução que segue do triplo $\{\phi\}\texttt{new}(p)\{\psi\}$.

   $$\frac{\models \phi \Rightarrow (\forall p'.p' \mapsto [-] \mathrel{-\!\!*} \psi[p'/p])}{\{\phi\}\texttt{new}(p)\{\psi\}}\ \text{Allocation}$$

4. Caso *lookup*:

   Suponhamos que $\models VCGen(\{\phi\}v{:=}p \rightarrow f\{\psi\})$, ou seja,

   $$\models VCGen(\{\phi\}v{:=}p \rightarrow f\{\psi\})$$
   $$\Leftrightarrow \models \phi \Rightarrow \exists v_1.(\psi[v_1/v] \wedge p \hookrightarrow [f : v_1])$$

   Assim,

   $$\frac{\models \phi \Rightarrow \exists v_1.(\psi[v_1/v] \wedge p \hookrightarrow [f : v_1])}{\{\phi\}v{:=}p \rightarrow f\{\psi\}}\ \text{Lookup}$$

   é uma dedução do triplo em questão.

5. Caso *while*:

   Suponhamos que $\models VCGen(\{\phi\}\texttt{while}\ b\ [\theta]\{\mathsf{C}\}\{\psi\})$, ou seja,

   $$\models VCGen(\{\phi\}\texttt{while}\ b\ [\theta]\{\mathsf{C}\}\{\psi\})$$
   $$\Leftrightarrow \models (\phi \Rightarrow \theta) \wedge ((\theta \wedge b) \Rightarrow wp(\mathsf{C},\theta)) \wedge ((\theta \wedge \neg b) \Rightarrow \psi) \wedge VC(\mathsf{C},\theta).$$

   É agora necessário eliminar as anotações do ciclo. Assim,

$$\frac{\models \phi \Rightarrow \theta \qquad \dfrac{D'}{\{(\theta \wedge b)\}\mathsf{C}\{\theta\}} \qquad \models (\theta \wedge \neg b) \Rightarrow \psi}{\{\phi\}\texttt{while}\ b\ \{\mathsf{C}\}\{\psi\}}$$

As condições laterais da derivação são válidas, por suposição. Falta apenas demonstrar que existe a derivação D'. De sabermos que $\models ((\theta \wedge b) \Rightarrow wp(\mathsf{C},\theta)) \wedge VC(\mathsf{C},\theta) \Leftrightarrow VCGen(\{(\theta \wedge b)\}\mathsf{C}\{\theta\})$ de acordo com a hipótese de indução vamos ter uma derivação em $\mathbf{S^{gb}}$ com conclusão $\{(\theta \wedge b)\}\mathsf{C}\{\theta\}$ onde as anotações dos ciclos *while* foram retiradas.

As provas dos outros casos relativos aos comandos da linguagem *While*, podem ser encontrados essencialmente em [24].

□

Mostraremos agora que a dedutibilidade de um triplo no sistema $\mathbf{S^{gb}}$ reflecte a validade de todas as fórmulas geradas pelo *VCGen*. Para tal, precisaremos de introduzir a noção de anotação dos ciclos *while* de um comando face a uma derivação.

Notação 4.3.6 Seja $\{\phi\}\mathsf{C}\{\psi\}$ um triplo de Hoare e seja D uma derivação deste triplo em $\mathbf{S^{gb}}$. A notação C^D representa o comando em que cada ciclo *while* é anotado com o correspondente invariante de ciclo na derivação D. Por exemplo, se D é uma derivação da forma:

$$\frac{\models \phi \Rightarrow \theta \qquad \dfrac{D'}{\{\theta \wedge b\}\mathsf{C}_0\{\theta\}} \qquad \models \theta \wedge \neg b \Rightarrow \psi}{\{\phi\}\texttt{while}\ b\ \{\mathsf{C}_0\}\{\psi\}}\ \text{While}$$

então $\mathsf{C}^D = \texttt{while}\ b\ [\theta]\{\mathsf{C}'\}$, onde $\mathsf{C}' = \mathsf{C}_0^{D'}$.

Teorema 4.3.7 Se $\vdash_{S^{gb}} \{\phi\}\mathsf{C}\{\psi\}$, então $\models VCGen(\{\phi\}\mathsf{C}^+\{\psi\})$, para algum comando C^+ (com ciclos *while* anotados) que resulte de C por anotação dos ciclos *while*.

Demonstração Mostraremos por indução em derivações de $\mathbf{S^{gb}}$ que: se D é uma derivação de $\{\phi\}\mathsf{C}\{\psi\}$ então $\models VCGen(\{\phi\}\mathsf{C}^D\{\psi\})$

1. Caso *mutation*:

$$\frac{\models \phi \Rightarrow (p \mapsto [-] * (p \mapsto [f:a] \mathrel{-\!\!*} \psi))}{\{\phi\}p{\rightarrow}f{:=}a\{\psi\}}\ \text{Mutation}$$

Queremos demonstrar que $VCGen(\{\phi\}p{\rightarrow}f{:=}a\{\psi\})$. (Note-se que a anotação de ciclos *while* não produz qualquer alteração neste comando). Ora,

$$\models VCGen(\{\phi\}p{\rightarrow}f{:=}a\{\psi\})$$
$$\Leftrightarrow \models \phi \Rightarrow (p \mapsto [-] * (p \mapsto [f : a] \mathrel{-\!\!*} \psi))$$

mas isto é precisamente a condição lateral imposta pela inferência *mutation* que assumimos existir.

2. Caso *dispose*:

$$\frac{\models \phi \Rightarrow (p \mapsto [-] * \psi)}{\{\phi\}\texttt{dispose}(p)\{\psi\}}\ \text{Dispose}$$

Queremos demonstrar que $VCGen(\{\phi\}\texttt{dispose}(p)\{\psi\})$. Ora,

$$\models VCGen(\{\phi\}\texttt{dispose}(p)\{\psi\})$$
$$\Leftrightarrow \models \phi \Rightarrow (p \mapsto [-] * \psi)$$

o que vem da suposição inicial, relativa à regra de *dispose* que estamos a tratar.

3. Caso *allocation*

$$\frac{\models \phi \Rightarrow (\forall p'.p' \mapsto [-] \mathrel{-\!\!*} \psi[p'/p])}{\{\phi\}\texttt{new}(p)\{\psi\}}\ \text{Allocation}$$

Queremos demonstrar que $VCGen(\{\phi\}\texttt{new}(p)\{\psi\})$. Ora,

$$\models VCGen(\{\phi\}\texttt{new}(p)\{\psi\})$$
$$\Leftrightarrow \models \phi \Rightarrow (\forall p'.p' \mapsto [-] \mathrel{-\!\!*} \psi[p'/p])$$

mas isto é a condição lateral de regra de *allocation*.

4. Caso *lookup*

$$\frac{\models \phi \Rightarrow \exists v_1.(\psi[v_1/v] \wedge p \hookrightarrow [f : v_1])}{\{\phi\}v{:=}p \rightarrow f\{\psi\}}\ \text{Lookup}$$

Queremos demonstrar que $VCGen(\{\phi\}v{:=}p \rightarrow f\{\psi\})$. Ora,

$$\models VCGen(\{\phi\}v{:=}p \rightarrow f\{\psi\})$$
$$\Leftrightarrow \models \phi \Rightarrow \exists v_1.(\psi[v_1/v] \wedge p \hookrightarrow [f : v_1])$$

e, tal como nos casos anteriores, esta validade é a condição lateral relativa à regra.

5. Caso *while*:

$$D = \frac{(\theta \wedge \neg b) \Rightarrow \psi \qquad \dfrac{D_0}{\{(\theta \wedge b)\}\mathsf{C}'\{\psi\}} \qquad \models \phi \Rightarrow \theta}{\{\phi\}\texttt{while}\ b\ \{\mathsf{C}'\}\{\psi\}}\ \text{While}$$

Teremos agora de anotar o comando de acordo com o invariante adequado. Ou seja, $(\texttt{while}\ b\ \{\mathsf{C}'\})^D = \texttt{while}\ b\ [\theta]\{\mathsf{C}_0\}$ onde $\mathsf{C}_0 = \mathsf{C}'^{D_0}$.

Queremos então demonstrar que

$$\begin{array}{l} \models VCGen(\{\phi\}\texttt{while}\ b\ [\theta]\{\mathsf{C}_0\}\{\psi\}) \\ \Leftrightarrow \models (\phi \Rightarrow \theta) \wedge wp(\mathsf{C}_0, \psi) \wedge ((\theta \wedge \neg b) \Rightarrow \psi) \wedge VC(\mathsf{C}_0, \psi) \end{array}$$

De sabermos que existe a derivação D temos em particular que $\models \phi \Rightarrow \theta$ e ainda que $\models \theta \wedge \neg b \Rightarrow \psi$.

De sabermos que existe a derivação D_0, por hipótese de indução temos que $\models VCGen(\{(\theta \wedge b)\}\mathsf{C}_0\{\theta\})$, isto é,$\models ((\theta \wedge b) \Rightarrow wp(\mathsf{C}_0, \psi)) \wedge (VC(\mathsf{C}_0, \psi))$ que era precisamente aquilo que faltava demonstrar.

Os restantes casos relativos aos comandos da linguagem *While*, podem essencialmente ser encontrados em [24].

□

Os resultados que vimos até agora permitem concluir a equivalência entre os vários sistemas formais e a validade das condições de verificação geradas pelo *VCGen*. Como já sabíamos que o sistema **S** é correcto em relação à semântica, podemos agora garantir a correcção do *VCGen*.

Teorema 4.3.8 Se $\models VCGen(\{\phi\}\mathsf{C}\{\psi\})$, então $\models \{\phi\}\mathsf{C}^-\{\psi\}$.

Demonstração Pelos teoremas já provados temos que:

1. Se $\models VCGen(\{\phi\}\mathsf{C}\{\psi\})$, então $\vdash_{S^{gb}} \{\phi\}\mathsf{C}^-\{\psi\}$. (Teorema 4.3.5)
2. Se $\vdash_{S^{gb}} \{\phi\}\mathsf{C}^-\{\psi\}$, então $\vdash_{S^{g}} \{\phi\}\mathsf{C}^-\{\psi\}$. (Teorema 4.2.1)
3. Se $\vdash_{S^{g}} \{\phi\}\mathsf{C}^-\{\psi\}$, então $\vdash_{S} \{\phi\}\mathsf{C}^-\{\psi\}$. (Teorema 4.1.1)
4. Se $\vdash_{S} \{\phi\}\mathsf{C}^-\{\psi\}$, então $\models \{\phi\}\mathsf{C}^-\{\psi\}$. (Teorema A.1.15)

Pela transitividade da implicação, fica provado que

$$\models VCGen(\{\phi\}\mathsf{C}\{\psi\}) \text{ implica } \models \{\phi\}\mathsf{C}^{-}\{\psi\}.$$

□

O resultado de completude do *VCGen* em relação ao sistema **S** é o seguinte:

Teorema 4.3.9 Se $\vdash_S \{\phi\}\mathsf{C}\{\psi\}$, então $\models VCGen(\{\phi\}\mathsf{C}^{+}\{\psi\})$, para algum comando C^{+} (com *while* anotados) que resulte de C por anotação dos comandos *while*.

Demonstração Pelos teoremas já provados temos que:

1. Se $\vdash_S \{\phi\}\mathsf{C}\{\psi\}$, então $\vdash_{S^g} \{\phi\}\mathsf{C}\{\psi\}$. (Teorema 4.1.2)

2. Se $\vdash_{S^g} \{\phi\}\mathsf{C}\{\psi\}$, então $\vdash_{S^{gb}} \{\phi\}\mathsf{C}\{\psi\}$. (Teorema 4.2.2)

3. Se $\vdash_{S^{gb}} \{\phi\}\mathsf{C}\{\psi\}$, então $\models VCGen(\{\phi\}\mathsf{C}^{+}\{\psi\})$. (Teorema 4.3.7)

Pela transitividade da implicação temos que:

$$\vdash_S \{\phi\}\mathsf{C}\{\psi\} \text{ implica } \models VCGen(\{\phi\}\mathsf{C}^{+}\{\psi\}).$$

□

No resultado anterior, note-se que C^{+} dependerá da derivação concreta considerada em $\mathbf{S^{gb}}$ e esta, por sua vez, será resultado das transformações sucessivas a uma derivação concreta considerada no sistema **S**.

A completude do *VCGen* em relação à semântica é parcial uma vez que os próprios sistemas formais não são completos em relação à semântica e uma vez que é necessário adivinhar invariantes de ciclo apropriados para anotar os comandos *while*. De facto, já para o fragmento da linguagem *While* apenas a completude parcial é garantida (veja-se em [3]).

No capítulo que se segue, apresenta-se uma implementação em Haskell de um animador do algoritmo de *VCGen* que foi aqui descrito.

Capítulo 5

Um protótipo do *VCGen*

No Capítulo 4 apresentou-se o algoritmo *VCGen* baseado na *weakest-precondition* para programas com ciclos anotados. Para consolidar a compreensão dos conceitos envolvidos, fizemos um pequeno protótipo em `Haskell` que anima esse algoritmo, e que pode ser obtido em `http://twiki.di.uminho.pt/twiki/pub/Research/CROSS/Tools/VCGEN.hs`.

Neste capítulo faremos a descrição desse protótipo, e das principais decisões de implementação. A listagem total do código está disponível em anexo B.

5.1 Implementação da linguagem de comandos e asserções

O *VCGen* é uma função que recebe um triplo de Hoare e produz o conjunto de condições de verificação que garante que esse triplo é derivável. Precisamos, por isso, de codificar a noção de triplo de Hoare e portanto teremos que definir uma forma de representar a linguagem de expressões (de cada tipo), a linguagem de comandos e a linguagem de asserções.

A sintaxe abstracta das expressões está descrita na Figura 2.1. Esta sintaxe está codificada em três tipos algébricos `ExpInt`, `ExpBool` e `ExpLoc`.

Dado que a tificação de variáveis não tem interferência no algoritmo *VCGen*, optamos por não nos preocuparmos em associar um tipo às variáveis. Cada variável é representada simplesmente por uma string do Haskell. Ainda assim, para além do tipo `Var` (que representa o conjunto $\mathbf{V}ar$), resolvemos declarar os tipos `VarI` e `VarL` para representar (ainda que artificialmente) os conjuntos $\mathbf{V}_{Int}$ e $\mathbf{V}_{Loc}$, respectivamente.

```
type Var  = String
type VarI = String
type VarL = String
```

Listing 5.1: Tipos de variáveis

Os tipos algébricos `ExpInt`, `ExpLoc` e `ExpBool` que a seguir se apresentam são usados para codificar expressões do tipo *Int*, *Loc* e *Bool*, respectivamente.

```
data ExpInt = Const  Int
            | X      VarI
            | Menos  ExpInt
            | Soma   ExpInt     ExpInt
            | Mult   ExpInt     ExpInt
            | Div    ExpInt     ExpInt
            | Mod    ExpInt     ExpInt
           deriving Eq

instance Show ExpInt where
    show  = prettyPrint
```

Listing 5.2: Expressões inteiras

```
data ExpBool= Top
            | Neg       ExpBool
            | E         ExpBool     ExpBool
            | Ou        ExpBool     ExpBool
            | Ig        ExpInt      ExpInt
            | Maior     ExpInt      ExpInt
            | Menor     ExpInt      ExpInt
            | MaiorIg   ExpInt      ExpInt
            | MenorIg   ExpInt      ExpInt
            | EqEnd     ExpLoc      ExpLoc
           deriving Eq

instance Show ExpBool where
    show = prettyPrintBool
```

Listing 5.3: Expressões booleanas

```
data ExpLoc = Nil
            | P         VarL
           deriving Eq

instance Show ExpLoc where
    show = prettyPrintLoc
```

Listing 5.4: Expressões endereço

A relação entre a codificação e a definição formal é evidente. Por exemplo, a expressão inteira $x+1$ será representada pela expressão em haskell `Soma (X "x")(Const 1)`.

A linguagem de comandos que está descrita na Figura 2.2 é codificada pelo tipo algébrico `Comando`

```
data Comando  = Skip
              | Atr       Var      Exp
              | Comp      Comando  Comando
              | If        ExpBool  Comando  Comando
              | While     ExpBool  Assert   Comando
              | Mutation  VarL     Field    Exp
              | New       VarL
              | Dispose   VarL
              | Lookup    VarI     VarL     Field
              | LookupF   VarL     VarL     Field
              deriving Eq

instance Show Comando where
    show = prettyPrintComando
```

Listing 5.5: Comandos

Como seria de esperar, cada construção da linguagem corresponde a um construtor do tipo. A excepção é o comando *lookup* que está dividido em dois casos: um para quando a variável para a qual estamos a copiar é do tipo *Int*, e o outro para quando é do tipo *Loc*.

Os comandos *lookup* e *mutation* recebem, entre outras coisas, o nome do campo dos *records* da *heap*. O nome dos campos dos *records* é simplesmente uma `string`.

```
type Field = String
```

Listing 5.6: Campos dos records

Há ainda uma outra diferença no que concerne à representação dos ciclos. Como vimos, os programas que serão passados ao *VCGen* têm os ciclos anotados com invariantes. Assim, o comando *while*, para além da expressão booleana que constitui a guarda do ciclo e do comando que constitui o corpo do ciclo, tem ainda um argumento do tipo `Assert` que é o invariante de ciclo.

`Assert` é o tipo das asserções cuja sintaxe se descreve na Figura 3.1. Como é natural, a implementação em Haskell das asserções faz-se através da declaração do tipo algébrico `Assert`.

```
data Assert= N      Assert
```

```
            | And    Assert      Assert
            | Or     Assert      Assert
            | Impl   Assert      Assert
            | Eq     Assert      Assert
            | QQ     Var         Assert
            | Ex     Var         Assert
            | Emp
            | CSep   Assert      Assert
            | ImpSep Assert      Assert
            | Map    VarL        Record
            | Exp    ExpBool
            deriving Eq

instance Show Assert where
    show = prettyPrintAsser
```

Listing 5.7: Asserções

Note-se que, para além das conectivas para a lógica da primeira ordem, e da inclusão das expressões booleanas como asserções, temos ainda as quatro conectivas especificas da lógica da separação *emp*, $\phi * \psi$, $\phi -\!\!* \psi$ e $p \mapsto r$. Dado que temos uma *heap* de *records*, temos que codificar também a noção de *record*. Isto é feito, muito simplesmente, com uma lista de pares com o nome do campo e uma expressão que denota o valor desse campo.

```
type Record = [(Field, Exp)]
```

Listing 5.8: Record

Note-se ainda que não incorporamos na linguagem de asserções a possibilidade de o utilizador definir novos predicados.

Estamos finalmente em condições de apresentar a codificação dos triplos de Hoare, pelo tipo algébrico `Triplo`.

```
data Triplo = T Assert Comando Assert
            deriving Show
```

Listing 5.9: Triplo de Hoare

5.2 Implementação dos algoritmos

O algoritmo de *VCGen* que apresentamos no capitulo anterior e que vamos implementar, baseia-se no cálculo da *weakest precondition*. A função `wp` que recebe um comando C e uma pós-condição ψ e produz uma asserção (a pre-condição mais fraca

de C face a ψ) está definida na Figura 4.3 . A implementação da função `wp` em Haskell foi feita da seguinte forma:

```
wp::(Comando,Assert)->Assert
wp  (Skip,psi)              = psi
wp  (Atr v a, psi)          = substAssert psi v a
wp  (Comp c1 c2,psi)        = wp(c1, wp(c2,psi))
wp  (If t ct cf, psi)       = And (Impl (Exp t) ( wp(ct,psi) ) ) (Impl (Exp (Neg t↩
    )) ( wp(cf,psi)) )
wp  (While b theta c,psi)   = theta
wp  (Dispose t ,psi)        = CSep psi (Map t [ ] )
wp  (New p,psi)             = let p1= (geraVar 0 "p '" (varsAssert psi ++ [p]) )
                              in (QQ p1 (ImpSep (Map  p1 [ ]) (substAssert psi p (↩
                                  L(P p1)) ) ) )
wp  (Mutation p f a ,psi)   = CSep (Map p [ ])(ImpSep (Map p [(f,a)] )(psi))
wp  (Lookup v p f,psi)      = let  p1= (geraVar 0 "v" (varsAssert psi ++[p]) )
                              in (Ex p1 (And (substAssert psi v (I (X p1)) ) (CSep↩
                                   (Map p [(f,(I (X p1)) )]) (Exp Top))))
wp  (LookupF v p f,psi)     = let  p1= (geraVar 0 "v" (varsAssert psi ++[p]) )
                              in (Ex p1 (And (substAssert psi v (L (P p1)) ) (CSep↩
                                   (Map p [(f,(L (P p1)) )]) (Exp Top))))
```

Listing 5.10: wp

Esta codificação segue de perto a definição. Note-se que esta codificação recorre às seguintes funções auxiliares: `substAssert` que implementa a substituição de variáveis nas asserções; `varsAssert` que colecciona a lista das variáveis que ocorrem numa asserção; e `geraVar` que produz uma variável fresca. Estas duas últimas funções são usadas para garantir uma correcta implementação da substituição (isto é, garantir que não há captura de variáveis).

A função `vc` que calcula as condições de verificação necessárias e suficientes para a derivabilidade do triplo $\{wp(\mathsf{C},\psi)\}\mathsf{C}\{\psi\}$ que está definida na Figura 4.4 foi implementado em Haskell pela seguinte função:

```
vc::(Comando,Assert)->[Assert]
vc  (Skip,psi)              = []
vc  (Atr p a, psi)          = []
vc  (New p , psi)           = []
vc  (Dispose a,psi)         = []
vc  (Mutation a b c, psi)   = []
vc  (Lookup v p f,psi)      = []
vc  (LookupF v p f,psi)     = []
vc  (Comp c1 c2, psi)       = norepetA (vc(c1,wp(c2,psi)))  (vc(c2,psi))
vc  (If b ct cf,psi)        = norepetA (vc(ct,psi)) (vc(cf,psi))
vc  (While b theta c,psi)   = norepetA (norepetA [Impl (And theta (Exp b)) (wp(c,↩
    psi)) ]  [Impl (And theta (N (Exp b))) psi])  (vc(c,theta))
```

Listing 5.11: VC

Esta implementação segue de perto a definição. Nos casos mais interessantes recorre à função `notrepetA` que implementa a reunião de conjuntos implementados como listas sem repetições. As condições produzidas pelo *VCGen* para um dado triplo $\{\phi\}\mathsf{C}\{\psi\}$ são então as condições produzidas por $vc(\mathsf{C},\psi)$ juntamente com a condição adicional de que $\phi \Rightarrow wp(\mathsf{C},\psi)$, conforme está definido na Figura 4.5. A função em Haskell que implementa o *VCGen* é a seguinte

```
vcgen :: Triplo -> [Assert]
vcgen (T phi c psi) =  norepetA [ Impl phi  (wp (c, ↩
   psi)) ] (vc (c, psi))
```

Listing 5.12: VCGen

5.3 Exemplos

Apresentaremos agora alguns exemplos de execução do *VCGen*.

Exemplo 5.3.1 Pelo exemplo de execução que segue podemos ver que as condições de verificação geradas pelo *VCGen* para o triplo $\{emp\}\texttt{New}(p)\{p \mapsto [-]\}$ são $emp \Rightarrow \forall p'(p' \mapsto [-] -\!\!* p' \mapsto [-])$. Com a ferramenta obtemos:

```
*Thesis> let phi= Emp
*Thesis> let psi = Map "p"[ ]
*Thesis> let c = (New "p")
*Thesis> vcgen(T phi c psi)
[( EMP => ( qualquer p'0.  ((p'0 |-> ) -* (p'0 |-> )))) ]
```

Exemplo 5.3.2 Vejamos agora que as condições de verificação geradas para o triplo $\{p \mapsto [-]\}p \rightarrow valor := 2\{p \mapsto [valor : 2]\}$ são $p \mapsto [-] \Rightarrow (p \mapsto [-] * (p \mapsto [valor : 2] -\!\!* p \mapsto [valor : 2]))$. As geradas pela ferramenta são:

```
*Thesis> let phi1=(Map "p"[ ])
*Thesis> let psi1=Map "p"[("valor",(I(Const 2)))]
*Thesis> let comando1= (Mutation "pvalor"(I(Const 2)))
*Thesis> vcgen (T phi1 comando1 psi1)
[( (p |-> ) => ((p |-> ) * ((p |-> (valor,2) ) -* (p |->
(valor,2) ))))]
```

Exemplo 5.3.3 Ilustremos agora as condições de verificação do triplo $\{p \mapsto [valor : 2]\} v := p \rightarrow valor \{p \mapsto [valor : 2] \wedge v = 2\}$

```
*Thesis> let phi2=Map "p"[("valor",(I(Const 2)))]
*Thesis> let psi2=And (Map "p"[("valor",(I(Const 2)))]) (Exp
(Ig(X "v") (Const 2)))
*Thesis> let c3= (Lookup "vpvalor")
*Thesis> vcgen (T phi2 c3 psi2)
[( (p |-> (valor,2) ) => ( exists v0.  (((p |-> (valor,2) ) &&
(v0 = 2)) && ((p |-> (valor,v0) ) * True))))]
```

Como se pode ver a condição gerada é $p \mapsto [valor : 2] \Rightarrow \exists v_1 . (p \mapsto [valor : 2] \wedge v_1 = 2 \wedge p \hookrightarrow [valor : v_1])$, coincide com as geradas pela ferramenta.

Exemplo 5.3.4 Para o triplo $\{p \mapsto [valor : 2] \wedge v := 2\}\texttt{dispose}(p)\{v := 2\}$

```
*Thesis> let phi3=And (Map "p"[("valor",(I(Const 2)))]) (Exp
(Ig(X "v") (Const 2)))
*Thesis> let c4=(Dispose "p")
*Thesis> let psi3=And (Map "p"[ ]) (Exp (Ig(X "v") (Const 2)))
*Thesis> vcgen (T phi3 c4 psi3)
[(((p |-> (valor,2) ) && (v = 2)) => ((EMP && (v = 2)) * (p |->
)))]
```

Como se pode ver a condição gerada é $p \mapsto [valor : 2] \wedge v = 2 \Rightarrow v = 2 * p \mapsto [-]$, coincide com a gerada pela ferramenta.

Exemplo 5.3.5 Recorde o seguinte triplo de Hoare:

$\{x = 1 \ \wedge \ y = 2\}$
$\texttt{new}(p)\texttt{;}$
$p \rightarrow \ valor := x\texttt{;}$
$x := y\texttt{;}$
$y := p \ \rightarrow \ valor\texttt{;}$
$\texttt{dispose}(p)\texttt{;}$
$\{x = 2 \ \wedge \ y = 1\}$

Cuja representação em Haskell seria a seguinte:

```
*Thesis> let tese1= And (Exp (Ig (X "x")(Const 2)) ) (Exp (Ig
(X "y") (Const 1)))
*Thesis> let tesecom1 = Comp (New "p")(Comp (Mutation "p
Valor"(I (X "x"))) (Comp (Atr "x"(I (X "y"))) (Comp (Lookup
"ypvalor") (Dispose "p"))))
*Thesis> let tese2= And (Exp (Ig (X "x")(Const 1)) ) (Exp (Ig
(X "y") (Const 2)))
```

Quando invocado o VCGen para esse triplo obtemos:

```
*Thesis> vcgen (T tese1 tesecom1 tese2) =
[( ((x = 2) && (y = 1)) =>
( qualquer p'0.  ((p'0 |-> ) -* ((p |-> )
* ((p |-> (Valor,x) ) -* ( exists v0.  ((((y = 1) && (v0 = 2))
* (p |-> )) && ((p |-> (valor,v0) ) * True)))))))))]
```

sendo aquilo que obtivemos no Exemplo 4.3.3.

Capítulo 6

Conclusões

6.1 Trabalho desenvolvido

Esta dissertação apresenta um estudo detalhado de uma metodologia de verificação de programas na linguagem $While_{pr}$, uma linguagem que estende a linguagem *While* com primitivas para o manuseamento de memória dinâmica. A metodologia assenta na lógica da separação e num algoritmo *VCGen* para gerar condições que garantam a correcção de um programa face a uma especificação. Em concreto, foi desenvolvido um *VCGen* para a variante da lógica da separação estabelecida para a linguagem $While_{pr}$. O *VCGen* apresentado baseia-se na estratégia da *weakest precondition* e trabalha sobre programas onde os ciclos While são anotados com um invariante de ciclo. O *VCGen* encontrado foi demonstrado adequado (correcto e completo) face ao sistema dedutivo $\mathbf{S^{gb}}$ que é uma versão *goal directed* do sistema base. Na literatura a que tivemos acesso, não encontramos trabalhos que procurem provar a correcção de um *VCGen* para a lógica da separação relativamente a um sistema formal. Um protótipo do algoritmo *VCGen* foi implementado na linguagem Haskell.

A definição da sintaxe e da semântica da linguagem $While_{pr}$ implicou certas escolhas específicas, nomeadamente ao nível da distinção entre expressões de inteiros e expressões de endereço e ao nível do tipo de células admitidas na *heap*. Embora a distinção entre inteiros e endereços limite a linguagem, não permitindo, por exemplo, aritmética de endereços, esta opção permite manter uma disciplina de tipos para as expressões base da linguagem, o que é desejável num contexto de verificação de programas. Na linguagem que definimos as células na *heap* limitam-se a *records* de tamanho fixo. A possibilidade de *records* com tamanhos diferenciados, embora conceptualmente não traga dificuldades adicionais, tornaria os detalhes deste estudo desnecessariamente complexos.

A lógica da separação foi introduzida na década de 90 e desde então tem existido inúmeros trabalhos nesta área. Contudo, para nós, foi surpreendente a complexidade dos detalhes necessários às definições da linguagem $While_{pr}$ e da lógica da separação associada e às provas de correcção e de relação entre os vários sistemas formais considerados para esta lógica. Um factor adicional de complexidade neste

trabalho relacionou-se com as opções específicas tomadas no desenho da linguagem $While_{pr}$. Embora aparentemente relativas a apenas pequenos detalhes, estas opções têm também um impacto considerável ao nível da formalização da lógica da separação associada. Na literatura a que pudemos aceder, muitos dos detalhes envolvidos em estudos da lógica da separação são frequentemente omitidos.

6.2 Trabalho futuro

A arquitectura de verificação de programas baseadas num *VCGen* tem usualmente duas fases. Na primeira fase são geradas as condições de verificação para um dado triplo. Numa segunda fase analisam-se as condições geradas, usualmente com recurso a ferramentas automáticas, no sentido de decidir acerca da validade das mesmas. A primeira fase é implementada à custa de um algoritmo de *VCGen*. O trabalho realizado nesta dissertação focou apenas nesta fase do processo de verificação. Uma continuidade natural para a dissertação seria desenvolver o estudo, no sentido de implementar também a segunda fase. Para tal, podem ser considerados dois caminhos distintos. Por um lado, a análise da validade das condições de verificação geradas (fórmulas da lógica da separação) pode ser feita directamente em termos da lógica da separação. Ferramentas como a biblioteca Ynot para o Coq ou o HOLfoot para Hol4 "implementam" a lógica da separação, permitindo o desenvolvimento interactivo de provas. Outro caminho passa por codificar asserções da lógica da separação geradas pelo *VCGen* em formulas da lógica de primeira ordem, uma abordagem seguida por exemplo em [10]. Este caminho implica, em particular, a representação adequada de fragmentos relevantes da lógica da separação em lógica de primeira ordem. A grande vantagem deste segundo caminho é o abrir a possibilidade de tirar partido das sofisticadas ferramentas existentes para decidir acerca de validade em lógica de primeira ordem.

O nosso algoritmo de *VCGen* é baseado na estratégia *weakest precondition*. Um caminho alternativo seria basear o *VCGen* na estratégia da *strongest postcondition*. Nesta estratégia, o caso da sequenciação de comandos começa por calcular a *strongest postcondition* para o primeiro comando, e essa condição é então usada para o cálculo da *strongest postcondition* para o comando seguinte. Seguindo um percurso semelhante ao usado para a estratégia *weakest precondition*, a primeira fase passaria por fazer evoluir o sistema **S** para um sistema *goal directed* (*forward*), onde as condições laterais das regras de inferência seriam distintas. Implementando um *VCGen* para a estratégia *strongest postcondition*, seria também interessante comparar os conjuntos de condições de verificação obtidos.

O protótipo de *VCGen* implementado em `Haskell`, à semelhança do estudo teórico desenvolvido, não prevê a possibilidade de definir novos predicados pelo utilizador, que possam ser usados nas anotações do código. Um enriquecimento importante do nosso protótipo seria o de prever esta possibilidade, o que permitiria, por exemplo, usar o *VCGen* para o caso de estudo das listas ligadas apresentado na secção 3.4. Um outro melhoramento, ainda ao nível da implementação, seria a criação de uma camada

de interface ao nível da recolha dos triplos, construindo um parser para a linguagem $While_{pr}$. Este tipo de facilidade foi implementando para a linguagem *While* simples em [24]. O código `Haskell` que implementa o *VCGen* foi desenvolvido sem grandes preocupações no que respeita à eficiência. Seria também útil considerar melhoramentos a este nível.

Bibliografia

[1] Holfoot, http://holfoot.heap-of-problems.org/, July 2012.

[2] The Coq Proof Assistant, http://coq.inria.fr/, July 2012.

[3] José Bacelar Almeida, Maria João Frade, Jorge Sousa Pinto, and Simão Melo de Sousa. *Rigorous Software Development: An Introduction to Program Verification*. Springer, 2011.

[4] Clark Barrett, Aaron Stump, and Cesare Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB). www.SMT-LIB.org, June 2012.

[5] Josh Berdine, Cristiano Calcagno, and Peter W. O'Hearn. Smallfoot: Modular automatic assertion checking with separation logic. In *In International Symposium on Formal Methods for Components and Objects*, pages 115–137. Springer, 2005.

[6] Josh Berdine, Cristiano Calcagno, and Peter W. O'Hearn. Smallfoot, http://www0.cs.ucl.ac.uk/staff/p.ohearn/smallfoot/, July 2012.

[7] Josh Berdine, Cristiano Calcagno, Peter W. O'Hearn, and Queen Mary. Symbolic execution with separation logic. In *In APLAS*, pages 52–68. Springer, 2005.

[8] Richard Bornat. Proving pointer programs in Hoare Logic, 2000.

[9] R. M. Burstall. Some Techniques for Proving Correctness of Programs which Alter Data Structures. In *Machine Intelligence 7*, pages 23–50. Edinburg University Press, 1972.

[10] Cristiano Calcagno and Matthew Hague. From separation logic to first-order logic. In *In FoSSaCs'05*, pages 395–409. Springer-Verlag, 2005.

[11] Adam Chlipala, Paul Govereau, Gregory Malecha, Greg Morrisett, Aleks Nanevski, Avi Shinnar, Matthieu Sozeau, and Ryan Wisnesky. The ynot, http://ynot.cs.harvard.edu/, July 2012.

[12] Dino Distefano and Matthew Parkinson. jStar: Bringing separation logic to Java, http://www.jstarverifier.org/, July 2012.

[13] C. Hoare. Procedures and parameters: An axiomatic approach. pages 102–116. 1971.

[14] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, October 1969.

[15] Samin Ishtiaq and Peter W. O'Hearn. BI as an Assertion Language for Mutable Data Structures. In *IN POPL*, pages 14–26, 2001.

[16] Élodie Jane Sims. *Pointer analysis and separation logic*. PhD thesis, University of Kansas State, 2007. page 35,42,43.

[17] Leonardo Moura and Nikolaj Bjørner. Formal methods: Foundations and applications. chapter Satisfiability Modulo Theories: An Appetizer, pages 23–36. Springer-Verlag, Berlin, Heidelberg, 2009.

[18] Aleksandar Nanevski, Greg Morrisett, Avraham Shinnar, Paul Govereau, and Lars Birkedal. Ynot: dependent types for imperative programs. *SIGPLAN Not.*, 43(9):229–240, September 2008.

[19] Peter O'Hearn, John Reynolds, and Hongseok Yang. Local Reasoning about Programs that Alter Data Structures, 2001.

[20] Peter W. O'Hearn and David J. Pym. The Logic of Bunched Implications. *BULLETIN OF SYMBOLIC LOGIC*, 5(2):215–244, 1999.

[21] John Reynolds. Separation Logic: A Logic for Shared Mutable Data Structures. pages 55–74. IEEE Computer Society, 2002.

[22] John C. Reynolds. Intuitionistic Reasoning about Shared Mutable Data Structure. In *Millennial Perspectives in Computer Science*, pages 303–321. Palgrave, 2000.

[23] John C. Reynolds. An introduction to Separation Logic, 2009.

[24] Isabel Santos and José Peixoto. Lógica de Hoare para uma linguagem imperativa simples Prova de correcção e implementação de um VCGen, http://twiki.di.uminho.pt/twiki/pub/research/cross/publications/mmc-pi-vcgen-report.pdf. Technical report, Departamento de Matemática, Universidade do Minho, 2011.

[25] Thomas Tuerk. A separation logic framework for HOL. Technical Report UCAM-CL-TR-799, University of Cambridge, Computer Laboratory, June 2011.

Anexo A

Provas

A.1 Resultados auxiliares

Lema A.1.1 $\models (\phi * (\phi \mathrel{-\!\!*} \psi)) \Rightarrow \psi$

Demonstração Queremos provar que para qualquer $s \in \mathbf{S}tate$ e $h \in \mathbf{H}eap$, $[\![(\phi * (\phi \mathrel{-\!\!*} \psi)) \Rightarrow \psi]\!](s,h)$, ou seja, se $[\![(\phi * (\phi \mathrel{-\!\!*} \psi)]\!](s,h)$ então $[\![\psi]\!](s,h)$.

Suponhamos que $[\![(\phi * (\phi \mathrel{-\!\!*} \psi)]\!](s,h)$, ou seja, $\exists\, h_0 \# h_1 \wedge h = h_0.h_1 \wedge [\![\phi]\!](s,h_0) \wedge [\![\phi \mathrel{-\!\!*} \psi]\!](s,h_1)$.

Como $[\![\phi]\!](s,h_0)$ e $h_0 \# h_1$, de $[\![\phi \mathrel{-\!\!*} \psi]\!](s,h_1)$ vem $[\![\psi]\!](s,h_0.h_1)$.

□

Lema A.1.2 $\models (\phi * \theta) \Rightarrow (\phi * (\psi \mathrel{-\!\!*} (\psi * \theta))$

Demonstração Suponhamos que $[\![\phi * \theta]\!](s,h) \Leftrightarrow [\![\phi]\!](s,h_1) \wedge [\![\theta]\!](s,h)$.

Queremos demonstrar que

$[\![\phi * (\psi \mathrel{-\!\!*} (\psi * \theta))]\!](s,h) \Leftrightarrow [\![\phi]\!](s,h_1) \wedge [\![(\psi \mathrel{-\!\!*} (\psi * \theta))]\!](s,h_2)$ A primeira é verdadeira por suposição inicial.

$[\![(\psi \mathrel{-\!\!*} (\psi * \theta))]\!](s,h_2)$. Suponhamos que $\forall h_3$: $h_3 \# h_1 \wedge [\![\psi]\!](s,h_3)$. Então

$[\![\psi * \theta]\!](s,h_3.h_2) \Leftrightarrow [\![\psi]\!](s,h_3) \wedge [\![\theta]\!](s,h_2)$ e ambos foram supostos verdadeiros inicialmente.

□

Lema A.1.3 Se $v \neq v_1 \wedge v \neq v_2 \models (v = m \wedge p \mapsto [f : a]) \Rightarrow \exists v_2.(v = m \wedge v_2 = a \wedge p \mapsto [f : v_2])$

Demonstração Suponhamos

$$
\begin{aligned}
&[\![v = m \wedge p \mapsto [f : a]]\!](s,h) \\
&\Leftrightarrow [\![v = m]\!](s,h) \wedge [\![p \mapsto [f : a]]\!](s,h) \\
&\Leftrightarrow [\![v]\!](s) = [\![m]\!](s) \wedge dom(h) = \{[\![p]\!](s)\} \wedge (h([\![p]\!](s)))(f) = [\![a]\!](s)
\end{aligned}
$$

Queremos demonstrar que

$[\![\exists v_2.(v = m \wedge v_2 = a \wedge p \mapsto [f : v_2])]\!](s,h)$
Tomando v_2 igual à a.
$\Leftrightarrow [\![v = m \wedge v_2 = a \wedge p \mapsto [f : v_2]]\!]([s|v_2 : a'],h)$
$\Leftrightarrow [\![v = m]\!]([s|v_2 : a'],h) \wedge [\![v_2 = a]\!]([s|v_2 : a'],h)$
$\wedge [\![p \mapsto [f : v_2]]\!]([s|v_2 : a'],h)$ para algum a'

Estas condições seguem da suposição inicial, tomando $a' = [\![a]\!](s)$.

□

Lema A.1.4 $\models \exists v'.(v' = m \wedge v = a \wedge p[v'/v] \mapsto [f : v]) \Rightarrow v = a \wedge p[m/v] \mapsto [f : a]$

Demonstração Suponhamos que

$[\![\exists v'.(v' = m \wedge v = a \wedge p[v'/v] \mapsto [f : v])]\!](s,h)$
$\Leftrightarrow [\![v' = m \wedge v = a \wedge p[v'/v] \mapsto [f : v]]\!]([s|v' : a'],h)$ para algum a'
$\Leftrightarrow [\![v' = m]\!]([s|v' : a'],h) \wedge [\![v = a]\!]([s|v' : a'],h) \wedge$
$[\![p[v'/v] \mapsto [f : v]]\!]([s|v' : a'],h)$ para algum a'

Queremos demonstrar que

$[\![v = a \wedge p[m/v] \mapsto [f : a]]\!](s,h)$
$\Leftrightarrow a)[\![v = a]\!](s,h) = T \wedge b)[\![p[m/v] \mapsto [f : a]]\!](s,h)$

a) Segue de $[\![v = a]\!]([s|v' : a'],h)$ uma vez que tomando $v' fresh$, $v' \neq v$ e $v' \notin free(a)$ b) De $[\![v' = m]\!]([s|v' : a'],h)$, sabemos que $[\![m]\!](s) = a'$ (recorde que $v' \notin free(m)$ por $v' fresh$) e sabíamos também $[\![p[v'/v] \mapsto [f : v]]\!]([s|v' : [\![m]\!](s)],h)$. Daqui segue $[\![p[m/v] \mapsto [f : v]]\!](s,h)$ como pretendido.

□

Lema A.1.5 $\models \exists v''.(v = m \wedge v'' = a \wedge p \mapsto [f : v'']) \Rightarrow \exists v''.(p \mapsto [f : v''] * (v = m \wedge v'' = a \wedge emp))$

Demonstração Suponhamos que

$[\![\exists v''.(v = m \wedge v'' = a \wedge p \mapsto [f : v''])]\!](s,h)$

Queremos demonstrar que

$[\![\exists v''.(p \mapsto [f : v''] * (v = m \wedge v'' = a \wedge emp))]\!](s,h)$
$\Leftrightarrow [\![p \mapsto [f : v''] * (v = m \wedge v'' = a \wedge emp)]\!]([s|v'' : a''], h)$ para algum a''
$\exists h_1 \# h_2 \wedge h = h_1.h_2 \wedge [\![p \mapsto [f : v'']]\!]([s|v'' : a''], h_1) \wedge$
$[\![(v = m \wedge v'' = a \wedge emp)]\!]([s|v'' : a''], h_2)$ para algum a''
$\Leftrightarrow [\![p \mapsto [f : v'']]\!]([s|v'' : a''], h_1) \wedge [\![v = m]\!]([s|v'' : a''], h_2) \wedge$
$[\![v'' = a]\!]([s|v'' : a''], h_2)$ para algum a''

Tomando $a'' = a'$, $h_1 = h$ $h_2 = [\,]$, as três condições seguem da suposição inicial.

□

Lema A.1.6 $\models \exists v'.(p[v'/v] \mapsto [f : v] * (v' = m \wedge v = a \wedge emp)) \Rightarrow \exists v'.(v' = m \wedge v = a \wedge p[v'/v] \mapsto [f : v])$

Demonstração Suponhamos que

$[\![\exists v'.(p[v'/v] \mapsto [f : v] * (v' = m \wedge v = a \wedge emp))]\!](s)h$
$\Leftrightarrow [\![p[v'/v] \mapsto [f : v]]\!]([s|v' : a'], h) \wedge$
$[\![v' = m]\!]([s|v' : a'], [\,]) \wedge [\![v = a]\!]([s|v' : v''], [\,])$ para algum a'.

Queremos demonstrar que

$[\![\exists v'.(v' = m \wedge v = a \wedge p[v'/v] \mapsto [f : v])]\!](s,h)$
$\Leftrightarrow [\![v' = m]\!]([s|v' : a'], h) \wedge [\![v = a]\!]([s|v' : a'], h) \wedge$
$[\![p[v'/v] \mapsto [f : v]]\!]([s|v' : a'], h)$ para algum a'

Tomando $a'' = a$, as três condições seguem da suposição inicial.

□

Lema A.1.7 $\models \exists v_1.(\psi[v_1/v] \wedge p \hookrightarrow [f : v_1]) \Rightarrow \exists v_1.((p \mapsto [f : v_1]) * (p \mapsto [f : v_1] \mathrel{-\!\!*} \psi[v_1/v]))$

Demonstração Suponhamos que

$$
\begin{array}{l}
[\![\exists v_1.(\psi[v_1/v] \wedge p \hookrightarrow [f : v_1])]\!](s,h) \\
\Leftrightarrow [\![\psi]\!]([s|v_1 : a'|v : a'],h) \wedge [\![p \mapsto [f : v_1] * \top]\!]([s|v_1 : a'],h) \text{ para algum } a'. \\
\Leftrightarrow \exists h_1,h_2 : h_1 \# h_2 \wedge h = h_1.h_2 \wedge [\![p \mapsto [f : v_1]]\!]([s|v_1 : a'],h_1) \wedge \\
[\![\top]\!]([s|v_1 : a'],h_2) \wedge [\![\psi]\!]([s|v_1 : a'|v : a'],h) \text{ para algum } a'
\end{array}
$$

Queremos demonstrar que

$$
\begin{array}{l}
[\![\exists v_1.((p \mapsto [f : v_1]) * (p \mapsto [f : v_1] \mathrel{-\!\!*} \psi[v_1/v]))]\!](s,h) \\
\Leftrightarrow \exists h'_1,h'_2 : h_1 \# h_2 \wedge h = h'_1.h'_2 \wedge [\![p \mapsto [f : v_1]]\!]([s|v_1 : a''],h'_1) \wedge \\
[\![(p \mapsto [f : v_1] \mathrel{-\!\!*} \psi[v_1/v])]\!]([s|v_1 : a''],h'_2) \text{ para algum } a''. \\
\text{Escolhendo } a'' = a',\ h'_1 = h_1 \text{ e } h'_2 = h_2 \text{ tem se, de facto:}
\end{array}
$$

1. $[\![p \mapsto [f : v_1]]\!]([s|v_1 : a'],h'_1)$ foi suposto como verdadeiro.
2. $[\![(p \mapsto [f : v_1] \mathrel{-\!\!*} \psi[v_1/v])]\!]([s|v_1 : a''],h_2)$ para algum a'', pois para todo h''_1 tal que $h''_1 \# h_2$ e $[\![p \mapsto [f : v_1]]\!]([s|v_1 : a''],h''_1)$ implica que $h''_1 = h_1$ e sabem que $[\![\psi[v_1/v]]\!]([s|v_1 : a''],h_2.h_1)$, que segue de $[\![\psi]\!]([s|v_1 : a''|v : a'],h)$.

□

Lema A.1.8 $\models \exists v.(\phi * (\phi \mathrel{-\!\!*} \psi)) \Rightarrow \exists v.\psi$

Demonstração Suponhamos que

$$
\begin{array}{l}
[\![\exists v.(\phi * (\phi \mathrel{-\!\!*} \psi))]\!](s,h) \\
\Leftrightarrow [\![\phi * (\phi \mathrel{-\!\!*} \psi)]\!]([s|v : a],h) \\
\Leftrightarrow \exists h_1,h_2 : h_1 \# h_2 \wedge h = h_1.h_2 \wedge [\![\phi]\!]([s|v : a],h_1) \wedge [\![\phi \mathrel{-\!\!*} \psi]\!]([s|v : a],h_2) \\
\Leftrightarrow [\![\phi]\!]([s|v : a],h_1) \wedge \forall h'_1 :\ h'_1 \# h_2 \wedge [\![\phi]\!]([s|v : a],h'_1) \Rightarrow \\
[\![\psi]\!]([s|v : a],h'_1.h_2)
\end{array}
$$

Queremos demonstrar que

$$
\begin{array}{l}
[\![\exists v.\psi]\!](s,h) \\
\Leftrightarrow [\![\psi]\!]([s|v : a],h)
\end{array}
$$

segue das duas suposições iniciais.

□

Lema A.1.9 $\models \phi \Rightarrow (\theta * (\gamma -\!\!* \psi))$ *e* $\models \psi \Rightarrow \psi'$ *então* $\models \phi \Rightarrow (\theta * (\gamma -\!\!* \psi'))$

Demonstração Queremos demonstrar que $[\![\phi \Rightarrow (\theta * (\gamma -\!\!* \psi'))]\!](s,h)$. Suponhamos que $[\![\phi]\!](s,h)$ queremos demonstrar que $[\![\theta * (\gamma -\!\!* \psi')]\!](s,h)$
$\Leftrightarrow \exists h_1, h_2$: $h_1 \# h_2 \ \wedge h = h_1.h_2 \wedge [\![\phi]\!](s,h_1) \wedge [\![\gamma -\!\!* \psi']\!](s,h_2)$.

De $[\![\phi]\!](s,h)$, pela hipótese, segue $[\![\theta * (\gamma -\!\!* \psi)]\!](s,h)$. Daqui, segue que $\exists h'_1, h'_2$: $h'_1 \# h_2 \wedge h = h'_1.h'_2 \wedge [\![\theta]\!](s,h'_1) \wedge [\![\gamma -\!\!* \psi]\!](s,h'_2)$. Tomando $h'_1 = h_1$ e $h'_2 = h_2$ segue o resultado pretendido. De facto, $[\![\theta]\!](s,h_1)$ é imediato e $[\![\gamma -\!\!* \psi']\!](s,h_2)$ é consequência de $[\![\gamma -\!\!* \psi]\!](s,h_2)$ e $\models \psi \Rightarrow \psi'$.

□

Lema A.1.10 $\models \phi \Rightarrow (\theta * \psi)$ e $\models \psi \Rightarrow \psi'$ então $\models \phi \Rightarrow (\theta * \psi')$

Demonstração Segue ideias análogas à demonstração do Lema A.1.9, embora a demonstração seja mais simples.

□

Lema A.1.11 $\models \phi' \Rightarrow \forall p'.(p' \mapsto [-] * \psi[p'/p])$ e $\models \psi \Rightarrow \psi'$ então $\models \phi' \Rightarrow \forall p'.(p' \mapsto [-] * \psi'[p'/p])$.

Demonstração Suponhamos que $[\![\phi' \Rightarrow (\forall p'.p' \mapsto [-] * \psi[p'/p])]\!](s,h)$. Suponhamos que $[\![\phi']\!](s,h)$ então $[\![\forall p'.(p' \mapsto [-] * \psi[p'/p])]\!](s,h) \Leftrightarrow [\![p' \mapsto [-] * \psi[p'/p]]\!]([s|p' : l],h)$ para todo $l \Leftrightarrow \exists h_1, h_2\ h_1 \# h_2 \wedge h = h_1.h_2 \wedge [\![p' \mapsto [-]]\!]([s|p' : l],h_1) \wedge [\![\psi]\!]([s|p' : l|p : l],h_2)$ para algum l.

Assim, com $\models \psi \Rightarrow \psi'$, também $[\![\psi]\!](s|p' : l|p : l],h_2)$, donde se conclui que $[\![\phi \Rightarrow \forall p'.(p' \mapsto [-] * \psi'[p'/p])]\!](s,h)$.

□

Lema A.1.12 $\models \phi' \Rightarrow \exists v_1.(\psi[v_1/v] \wedge p \hookrightarrow [f : v_1])$ e $\models \psi \Rightarrow \psi'$ então $\models \phi' \Rightarrow \exists v_1.(\psi'[v_1/v] \wedge p \hookrightarrow [f : v_1])$.

Demonstração Segue por argumentos essencialmente análogos aos usados nas provas anteriores.

Lema A.1.13 $\models p \mapsto [-] * (\forall p'.p' \mapsto [-] -\!\!* \psi[p'/p]) \Rightarrow p \mapsto [-] * (p \mapsto [-] -\!\!* \psi)$

Demonstração Suponhamos que $[\![p \mapsto [-] * (\forall p'.p' \mapsto [-] -\!\!* \psi[p'/p])]\!](s,h)$
$\Leftrightarrow \exists h_1, h_2$: $h_1 \# h_2 \wedge h = h_1.h_2 \wedge [\![p \mapsto [-]]\!](s,h_1) \wedge [\![\forall p'.p' \mapsto [-] -\!\!* \psi[p'/p])]\!](s,h_2)$
$\Leftrightarrow dom(h_1) = \{[\![p]\!](s)\} \wedge [\![p' \mapsto [-] -\!\!* \psi[p'/p])]\!]([s|p' : l],h_2)$ para todo l

Queremos demonstrar que $[\![p \mapsto [-] * (p \mapsto [-] -\!\!* \psi)]\!](s,h)$
$\Leftrightarrow \exists h'_1, h'_2$: $h'_1 \# h'_2 \wedge h = h'_1.h'_2 \wedge [\![p \mapsto [-]]\!](s,h'_1) \wedge [\![p \mapsto [-] -\!\!* \psi]\!](s,h'_2)$.
Assim, basta particularizar l a $[\![p]\!](s)$ e tomar $h'_1 = h_1$ e $h'_2 = h_2$.

□

Lema A.1.14 Se $v \notin free(\mathsf{C})$ e $(\mathsf{C},(s,h)) \rightsquigarrow (\mathsf{C}',(s',h'))$ então $v \notin free(\mathsf{C}')$ e $(\mathsf{C},([s|v:a],h)) \rightsquigarrow (\mathsf{C}',([s'|v:a],h'))$ para todo o a.

Demonstração Por indução em $\rightsquigarrow$

□

Teorema A.1.15 (Corresponde ao Teorema 3.3.4) Se $\vdash_S \{\phi\}C\{\psi\}$ então $\models \{\phi\}C\{\psi\}$

Demonstração Por indução associado nas derivações de **S** : As provas dos casos relativos à manipulação da memória dinâmica, assim como a *frame* e eliminação de variáveis auxiliares encontram-se detalhadas em 3.3.4 Teorema 3.3.4.

1. Caso *skip*:
   Queremos demonstrar que $\models \{\phi\}\mathtt{skip}\{\phi\}$, ou seja, se $\forall(s,h) \in \mathbf{S}tate$ se $[\![\phi]\!](s,h)$ então

   (a) $safe(\mathtt{skip},(s,h))$

   (b) $\forall(s',h') \in \mathbf{S}tate$ se $(\mathtt{skip},(s,h)) \rightsquigarrow^* (\mathtt{skip},(s',h'))$ então $[\![\phi]\!](s',h')$

   (a) Comecemos por demonstrar que $safe(\mathtt{skip},(s,h))$. Ora $(\mathtt{skip},(s,h))$ é uma configuração terminal e portanto $(\mathtt{skip},(s,h)) \rightsquigarrow^* \gamma$ implica $\gamma = (\mathtt{skip},(s,h)) \neq abort$.

   (b) Suponhamos agora que $(\mathtt{skip},(s,h)) \rightsquigarrow^* (\mathtt{skip},(s,h))$.Então, pelas definições de $\rightsquigarrow$ e $\rightsquigarrow^*$ segue que $s' = s$ e $h' = h$. Assim, $[\![\phi]\!](s',h') \Leftrightarrow [\![\phi]\!](s,h)$ e esta última é verdadeira por suposição.

2. Caso atribuição:
   Queremos demonstrar que $\models \{\psi[a/v]\}v{:=}a\{\psi\}$, ou seja, $\forall(s,h) \in \mathbf{S}tate$ se $[\![\psi[a/v]]\!](s,h)$ então

   (a) $safe(v{:=}a,(s,h))$

   (b) $\forall(s',h') \in \mathbf{S}tate$ se $(v{:=}a,(s,h)) \rightsquigarrow^* (\mathtt{skip},(s',h'))$ então $[\![\psi]\!](s',h')$

   (a) Comecemos por demonstrar que $safe(v{:=}a,(s,h))$, ou seja, $(v{:=}a,(s,h)) \not\rightsquigarrow^* abort$. Atendendo a definição de $\rightsquigarrow$, a configuração *abort* nunca pode ser obtida a partir de uma atribuição.

   (b) Falta agora demonstrar que $\forall(s',h') \in \mathbf{S}tate$ se $(v{:=}a,(s,h)) \rightsquigarrow^* (\mathtt{skip},(s',h'))$ então $[\![\psi]\!](s',h')$. Da definição de $\rightsquigarrow$ e $\rightsquigarrow^*$, sabemos que

   $$(v{:=}a,(s,h)) \rightsquigarrow (\mathtt{skip},(s',h'))$$

   implica que $s' = [s|v : [\![a]\!](s)]$ e $h' = h$ Ora, $[\![\psi]\!](s',h') \Leftrightarrow [\![\psi]\!]([s|v : [\![a]\!](s)],h) \Leftrightarrow [\![\psi[a/v]]\!](s,h)$.

3. Caso composição:
   Queremos mostrar que $\models \{\phi\}\mathsf{C}_1;\mathsf{C}_2\{\psi\}$, assumindo como hipóteses que $\models \{\phi\}\mathsf{C}_1\{\theta\}$ e $\models \{\theta\}\mathsf{C}_2\{\psi\}$.

   Queremos então demonstrar que se $[\![\phi]\!](s,h)$ então

   (a) $safe(\mathsf{C}_1;\mathsf{C}_2,(s,h))$

   (b) $\forall(s',h') \in \mathbf{S}tate$, $(\mathsf{C}_1;\mathsf{C}_2,(s,h)) \rightsquigarrow^* (\mathtt{skip},(s',h'))$ então $[\![\psi]\!](s',h')$

   (a) Suponhamos que $[\![\phi]\!](s,h)$. Comecemos por demonstrar que $safe(\mathsf{C}_1;\mathsf{C}_2,(s,h))$, ou seja, $(\mathsf{C}_1;\mathsf{C}_2,(s,h)) \not\rightsquigarrow^* abort$. Suponhamos que não. Então, ou $(\mathsf{C}_1,(s,h)) \rightsquigarrow^* abort$ ou $(\mathsf{C}_1;\mathsf{C}_2,(s,h)) \rightsquigarrow^* (\mathtt{skip};\mathsf{C}_2,(s',h')) \rightsquigarrow (\mathsf{C}_2,(s',h')) \rightsquigarrow^* abort$.
   De $[\![\phi]\!](s,h)$, pela primeira hipótese de indução, $safe(\mathsf{C}_1,(s,h))$, o que implica que $(\mathsf{C}_1,(s,h)) \not\rightsquigarrow^* abort$.
   Uma vez que $(\mathsf{C}_1,(s,h)) \rightsquigarrow^* (\mathtt{skip},(s',h'))$, pela primeira hipótese de indução $[\![\theta]\!](s',h')$. Daqui, pela segunda hipótese de indução, segue $safe(\mathsf{C}_2,(s',h'))$ o que contraria $(\mathsf{C}_2,(s',h')) \rightsquigarrow^* abort$.

   (b) Falta agora demonstrar que se $(\mathsf{C}_1;\mathsf{C}_2,(s,h)) \rightsquigarrow^* (\mathtt{skip},(s',h'))$ então $[\![\psi]\!](s',h')$. A partir da suposição $(\mathsf{C}_1;\mathsf{C}_2,(s,h)) \rightsquigarrow^* (\mathtt{skip},(s',h'))$ prova-se que $\exists(s_2,h_2) \in \mathbf{S}tate$ tal que $(\mathsf{C}_1;\mathsf{C}_2,(s,h)) \rightsquigarrow^* (\mathtt{skip};\mathsf{C}_2,(s_2,h_2)) \rightsquigarrow (\mathsf{C}_2,(s_2,h_2)) \rightsquigarrow^* (\mathtt{skip},(s',h'))$.
   De sabermos que $[\![\phi]\!](s,h)$ e $(\mathsf{C}_1,(s,h)) \rightsquigarrow^* (\mathtt{skip},(s_2,h_2))$, temos pela primeira hipótese de indução $[\![\theta]\!](s_2,h_2)$.
   De sabermos que $[\![\theta]\!](s_2,h_2)$ e $(\mathsf{C}_2,(s_2,h_2)) \rightsquigarrow^* (\mathtt{skip},(s',h'))$ então pela segunda hipótese de indução, temos que $[\![\psi]\!](s',h')$, que era precisamente aquilo que pretendiamos demonstrar.

4. Caso *if*:

   Queremos demonstrar que $\models \{\phi\}\mathtt{if}\ b\ \mathtt{then}\ \mathsf{C}_t\ \mathtt{else}\ \mathsf{C}_f\{\psi\}$, assumindo como hipóteses de indução $\models \{\phi \wedge b\}\mathsf{C}_t\{\psi\}$ e $\models \{\phi \wedge \neg b\}\mathsf{C}_f\{\psi\}$.

   Suponhamos $[\![\phi]\!](s,h)$. Teremos que mostrar

   (a) $safe(\mathtt{if}\ b\ \mathtt{then}\ \mathsf{C}_t\ \mathtt{else}\ \mathsf{C}_f,(s,h))$

   (b) $\forall(s',h') \in \mathbf{S}tate$, se $(\mathtt{if}\ b\ \mathtt{then}\ \mathsf{C}_t\ \mathtt{else}\ \mathsf{C}_f,(s,h)) \rightsquigarrow^* (\mathtt{skip},(s',h'))$ então $[\![\psi]\!](s',h')$

   Partiremos a prova em dois casos, consoante o valor de $[\![b]\!](s,h)$. Consideremos apenas o caso de $[\![b]\!](s,h)$ (o caso $\neg[\![b]\!](s,h)$ tem demonstração análoga).

   Caso $[\![b]\!](s,h)$:

   (a) Comecemos por demonstrar que $safe(\mathtt{if}\ b\ \mathtt{then}\ \mathsf{C}_t\ \mathtt{else}\ \mathsf{C}_f,(s,h))$, isto é, $(\mathtt{if}\ b\ \mathtt{then}\ \mathsf{C}_t\ \mathtt{else}\ \mathsf{C}_f,(s,h)) \not\rightsquigarrow^* abort$. Suponhamos que não. Tendo em conta a relação $\rightsquigarrow$ e da suposição$[\![b]\!](s,h)$, temos que

$(\texttt{if } b \texttt{ then } \mathsf{C}_t \texttt{ else } \mathsf{C}_f, (s,h)) \rightsquigarrow (\mathsf{C}_t, (s,h)) \not\rightsquigarrow^* abort$. Mas, de sabermos que $[\![\phi]\!](s,h)$ e $[\![b]\!](s,h)$, temos então que $[\![\phi \wedge b]\!](s,h)$ e, pela hipótese de indução, temos que $safe(\mathsf{C}_t, (s,h))$, o que contradiz $(\mathsf{C}_t, (s,h)) \rightsquigarrow^* abort$.

(b) Falta agora demonstrar que
$\forall (s',h') \in \mathbf{S}tate$ se $(\texttt{if } b \texttt{ then } \mathsf{C}_t \texttt{ else } \mathsf{C}_f, (s,h)) \rightsquigarrow^* (\texttt{skip}, (s',h'))$ então $[\![\psi]\!](s',h')$. Atendendo às definições de $\rightsquigarrow$ e $\rightsquigarrow^*$ e ao caso em que estamos, sabemos que

$(\texttt{if } b \texttt{ then } \mathsf{C}_t \texttt{ else } \mathsf{C}_f, (s,h)) \rightsquigarrow (C_t, (s,h)) \rightsquigarrow^* (\texttt{skip}, (s',h'))$. Assim, de termos $[\![\phi \wedge b]\!](s,h)$ e $(\mathsf{C}_t, (s,h)) \rightsquigarrow^* (\texttt{skip}, (s',h'))$, pela primeira hipótese de indução, segue que $[\![\psi]\!](s',h')$.

5. Caso conseq. :
Suponhamos que $\models \phi' \rightarrow \phi$, $\models \psi \rightarrow \psi'$ e pela hipótese de indução, suponhamos ainda que $\models \{\phi\}\mathsf{C}\{\psi\}$

Queremos demonstrar que se $[\![\phi']\!](s,h) = T$ então

(a) $safe(C,(s,h))$

(b) $\forall (s',h') \in \mathbf{S}tate$ se $(C,(s,h)) \rightsquigarrow (\texttt{skip}, (s',h'))$ então $[\![\psi]\!](s',h')$.

(a) Comecemos por demonstrar que $safe(\mathsf{C},(s,h))$. Ora, de sabermos que $[\![\phi']\!](s,h)$ e de sabermos que $\models \phi' \Rightarrow \phi$, então $[\![\phi]\!](s,h)$. Daqui, pela hipótese de indução, temos que $safe(\mathsf{C},(s,h))$.

(b) Falta agora provar que se $(C,(s,h)) \rightsquigarrow (\texttt{skip}, (s',h'))$, então $[\![\psi']\!](s',h')$. De $(C,(s,h)) \rightsquigarrow (\texttt{skip}, (s',h'))$, por hipótese de indução, temos que $[\![\psi]\!](s',h')$ como temos ainda que $\models \psi' \rightarrow \psi$, $[\![\psi']\!](s',h')$.

□

Lema A.1.16 Quando $safe(\mathsf{C},(s,h_0))$ e $(\mathsf{C},(s,h_0.h_1)) \rightsquigarrow^* (\texttt{skip}, (s',h'))$ então $\exists\, h'_0$ tal que $(\mathsf{C},(s,h_0)) \rightsquigarrow^* (\texttt{skip}, (s',h'_0))$ e $h' = h'_0.h_1$.

Demonstração Por indução no comprimento da sequência de transição $\rightsquigarrow^*$.

- Caso 0 passos:
Então $\mathsf{C} = \texttt{skip}$, $s' = s$ e $h' = h_0.h_1$. Logo, basta tomar $h'_0 = h_0$.

- Caso n+1 passos:

Suponhamos $safe(\mathsf{C},(s,h_0))$ e ainda $(\mathsf{C},(s,h_0.h_1)) \rightsquigarrow^{n+1} (\texttt{skip}, (s',h'))$.

Queremos demonstrar que $\exists\, h''_0 : (\mathsf{C},(s,h_0)) \rightsquigarrow^* (\mathsf{C}', (\texttt{skip}, h''_0))$ e $h' = h''_0.h_1$.
Ora,

$$(\mathsf{C},(s,h_0.h_1)) \rightsquigarrow (\mathsf{C}',(s'',h'')) \rightsquigarrow^n (\texttt{skip},(s',h')) \text{ para algum } \mathsf{C}',s'',h''$$

Por $safe(\mathsf{C},(s,h_0))$ e por $(\mathsf{C},(s,h_0.h_1)) \rightsquigarrow (\mathsf{C}',(s'',h''))$, atendendo à definição de $\rightsquigarrow$, segue que $h'' = h_0'''.h_1$ e $(\mathsf{C},(s,h_0)) \rightsquigarrow (\mathsf{C}',(s',h_0'''))$ para algum h_0'''. Daqui e de $(\mathsf{C}',(s'',h'')) \rightsquigarrow^n (\texttt{skip},(s,h'))$, por hipótese de indução existe h_0'' nas condições pretendidas.

□

Lema A.1.17 Se $safe(\mathsf{C},(s,h))$ e $h\#h'$ então $safe(C,(s,h.h'))$

Demonstração Suponhamos que não temos $safe(\mathsf{C},(s,h.h'))$. Então, existe uma computação $(\mathsf{C},(s,h.h')) \rightsquigarrow^* abort$.

Retirando h' da sequência finita de transições da computação resulta em $(\mathsf{C},(s,h)) \rightsquigarrow abort$, o que é absurdo, pois supusemos que $safe(\mathsf{C},(s,h))$. Logo, o absurdo resultou de termos suposto que não é $safe(\mathsf{C},(s,h.h'))$ e portanto temos que $safe(\mathsf{C},(s,h.h'))$.

□

Proposição A.1.18 (Corresponde a Proposição 4.1.3)

Se $\vdash_{S^g} \{\phi\}\mathsf{C}\{\psi\}$ e $modifies(\mathsf{C}) \cap free(\theta) = \varnothing$ então $\vdash_{S^g} \{\phi * \theta\}\mathsf{C}\{\psi * \theta\}$.

Demonstração Por indução nas derivações de $\mathbf{S^g}$:

As provas dos casos relativos a manipulação de memória dinâmica já foram apresentados no Capítulo 4, em particular na Proposição 4.1.3. Aqui detalham-se as provas dos restantes casos.

1. Caso *skip*:

$$\frac{}{\{\phi\}\texttt{skip}\{\phi\}}\ \text{Skip}$$

Queremos demonstrar que se $modifies(\texttt{skip}) \cap free(\theta) = \varnothing$ então $\vdash_{S^g} \{\phi * \theta\}\texttt{skip}\{\phi * \theta\}$.

Para tal, basta então tomar a derivação:

$$\frac{}{\{\phi * \theta\}\texttt{skip}\{\phi * \theta\}}\ \text{Skip}$$

2. Caso atribuição:

$$\frac{}{\{\psi[a/v]\}v{:=}a\{\psi\}}\ \text{Atrib.}$$

Queremos demonstrar que se $modifies(v{:=}a) \cap free(\theta) = \varnothing$ então $\vdash_{S^g} \{\psi[a/v] * \theta\} v{:=}a \{\psi * \theta\}$.

Basta então tomar,

$$\frac{}{\{(\psi * \theta)[a/v]\} v{:=}a \{\psi * \theta\}} \text{ Atrib.}$$

Note-se que, como $modifies(v{:=}a) \cap free(\theta) = \varnothing$, $v \notin free(\theta)$ e, portanto, $(\psi * \theta)[a/v] = \psi[a/v] * \theta$.

3. Caso *if*:

$$\frac{\dfrac{D_1}{\{\phi \wedge b\}\mathsf{C}_t\{\psi\}} \quad \dfrac{D_2}{\{\phi \wedge \neg b\}\mathsf{C}_f\{\psi\}}}{\{\phi\}\texttt{if } b \texttt{ then } \mathsf{C}_t \texttt{ else } \mathsf{C}_f\{\psi\}} \text{ If}$$

Queremos demonstrar que se $modifies(\texttt{if } b \texttt{ then } \mathsf{C}_t \texttt{ else } \mathsf{C}_f) \cap free(\theta) = \varnothing$ então $\vdash_{S^g} \{\phi * \theta\}\texttt{if } b \texttt{ then } \mathsf{C}_t \texttt{ else } \mathsf{C}_f\{\psi * \theta\}$.

De sabermos que $modifies(\texttt{if } b \texttt{ then } \mathsf{C}_t \texttt{ else } \mathsf{C}_f) \cap free(\theta) = \varnothing$ sabemos em particular que $modifies(\mathsf{C}_t) \cap free(\theta) = \varnothing$, e $modifies(\mathsf{C}_f) \cap free(\theta) = \varnothing$. Aplicando as hipóteses de indução associados a D_1 e D_2, segue então que existem derivações D_1' e D_2' de $\{(\phi \wedge b) * \theta\}\mathsf{C}_t\{\psi\}$ e $\{(\phi \wedge \neg b) * \theta\}\mathsf{C}_f\{\psi\}$. Note-se que, de facto, $(\phi \wedge b) * \theta \Leftrightarrow (\phi * \theta) \wedge b$, pois a avaliação das expressões booleanas não dependem da heap.

Assim, basta então tomar,

$$\frac{\dfrac{D_1'}{\{(\phi * \theta) \wedge b\}\mathsf{C}_t\{\psi * \theta\}} \quad \dfrac{D_2'}{\{(\phi * \theta) \wedge \neg b\}\mathsf{C}_f\{\psi * \theta\}}}{\{\phi * \theta\}\texttt{if } b \texttt{ then } \mathsf{C}_t \texttt{ else } \mathsf{C}_f\{\psi * \theta\}} \text{ If}$$

4. Caso *while*:

$$\frac{\dfrac{D_1}{\{\theta' \wedge b\}\mathsf{C}\{\theta'\}}}{\{\theta'\}\texttt{while } b \{\mathsf{C}\}\{\theta' \wedge \neg b\}} \text{ While}$$

Queremos demonstrar que se $modifies(\texttt{while } b \{\mathsf{C}\}) \cap free(\theta) = \varnothing$ então $\vdash_{S^g} \{\theta' * \theta\}\texttt{while b } \{\mathsf{C}\}\{(\theta' \wedge \neg \mathsf{b}) * \theta\}$.

De sabermos que $modifies(\texttt{while } b \{\mathsf{C}\}) \cap free(\theta) = \varnothing$, sabemos em particular que $modifies(\mathsf{C}) \cap free(\theta) = \varnothing$. Daqui e da hipótese de indução, temos que existe derivação em $\mathbf{S^g}$ com conclusão $\{(\theta' \wedge b) * \theta\}\mathsf{C}\{\theta' * \theta\}$. Como $(\theta' \wedge b) * \theta \Leftrightarrow (\theta' * \theta) \wedge b$, podemos usar uma inferência de consequência seguida de uma inferência de *while* para construir a seguinte derivação:

$$\dfrac{\dfrac{\dfrac{D'_1}{\{(\theta' \wedge b) * \theta\}\mathsf{C}\{\theta' * \theta\}}\text{Conseq.}}{\{(\theta' * \theta) \wedge b\}\mathsf{C}\{\theta' * \theta\}}}{\{\theta' * \theta\}\texttt{while b do}\{\mathsf{C}\}\{(\theta' * \theta) \wedge \neg \texttt{b}\}}\text{While}$$

5. Caso consequência :

$$\dfrac{\models \phi \Rightarrow \phi' \qquad \dfrac{D_1}{\{\phi\}\mathsf{C}\{\psi\}} \qquad \models \psi \Rightarrow \psi'}{\{\phi'\}\mathsf{C}\{\psi'\}}\text{Conseq.}$$

Queremos demonstrar que se $modifies(\mathsf{C}) \cap free(\theta) = \varnothing$ então $\vdash_{S^g} \{\phi' * \theta\}\mathsf{C}\{\psi' * \theta\}$.

De sabermos que $modifies(\mathsf{C}) \cap free(\theta) = \varnothing$ e pela hipótese de indução segue que existe uma derivação D'_1 de $\{\phi * \theta\}\mathsf{C}\{\psi * \theta\}$.

Assim, basta então tomar,

$$\dfrac{\dfrac{D'_1}{\{\phi * \theta\}\mathsf{C}\{\psi * \theta\}}\text{Conseq.}}{\{\phi' * \theta\}\mathsf{C}\{\psi' * \theta\}}$$

Falta apenas argumentar acerca da validade das condições necessárias à inferência da consequência, isto é, provar $\models \phi * \theta \Rightarrow \phi' * \theta$ e $\models \psi' * \theta \Rightarrow \psi * \theta$. Ora, suponhamos que $[\![\phi * \theta]\!](s,h)$, ou seja, $\exists h_1, h_2$ tais que $h = h_1.h_2$ e $h_1 \# h_2$ e $[\![\phi]\!](s,h_1)$ e $[\![\theta]\!](s,h_2)$.

Queremos demonstrar que $[\![\phi' * \theta]\!](s,h)$, ou seja, temos que demonstrar que $\exists h_3, h_4$ tais que $h = h_3.h_4$, $h_3 \# h_4$, $[\![\phi']\!](s,h_3)$ e $[\![\theta]\!](s,h_4)$. Tomemos $h_3 = h_1$ e $h_4 = h_2$. Assim, $[\![\theta]\!](s,h_4)$(pois $[\![\theta]\!](s,h_2)$) e de sabermos que $\models \phi \Rightarrow \phi'$ e $[\![\phi]\!](s,h_1)$temos que $[\![\phi']\!](s,h_1)$ e portanto $[\![\phi]\!](s,h_1)$, ou seja, $[\![\phi]\!](s,h_3)$.

Uma vez que $\models \psi' \Rightarrow \psi$, a validade de $\psi' * \theta \Rightarrow \psi * \theta$ segue por razões análogas.

□

Proposição A.1.19 (Corresponde à Proposição 4.1.4) Se $\vdash_{S^g} \{\phi\}\mathsf{C}\{\psi\}$ e $v \notin free(\mathsf{C})$ então $\vdash_{S^g} \{\exists v.\phi\}\mathsf{C}\{\exists v.\psi\}$

Demonstração Por indução nas deduções de $\mathbf{S^g}$.

As provas dos casos relativos à manipulação de memória já foram apresentadas no Capítulo 4, na Proposição 4.1.4. Aqui detalham-se as provas dos restantes casos.

1. Caso *skip*:

$$\frac{}{\{\phi\}\texttt{skip}\{\phi\}}\ \text{Skip}$$

Queremos demonstrar que se $v \notin free(\texttt{skip})$ então $\vdash_{Sg} \{\exists v.\phi\}\texttt{skip}\{\exists v.\phi\}$. Ora,

$$\frac{}{\{\exists v.\phi\}\texttt{skip}\{\exists v.\phi\}}\ \texttt{skip}$$

é uma derivação com o formato pretendido.

2. Caso atribuição:

$$\frac{}{\{\psi[a/x]\}x := a\{\psi\}}\ \text{Atrib.}$$

Queremos demonstrar que $v \notin free(x := a)$ então $\vdash_{Sg} \{\exists v.\psi[a/x]\}x := a\{\psi\}$. Ora, basta então tomar,

$$\frac{\dfrac{}{\{(\exists v.\psi)[a/x]\}x := a\{\exists v.\psi\}}\ \text{Atrib.}}{\{\exists v.\psi[a/x]\}x := a\{\exists v.\psi\}}\ \text{Conseq.}$$

Teremos agora de provar que $\models \exists v.\psi[a/x] \Rightarrow (\exists v.\psi)[a/x]$ e ainda que $\models \exists v.\psi \Rightarrow \exists v.\psi$. A segunda é trivialmente verdadeira e a primeira segue de $v \notin free(x := a) \Leftrightarrow v \notin \{x\} \cup \mathbf{V}ar(a)$, pelo que podemos então passar a substituição para o exterior da quantificação. Na verdade, a aplicação da Conseq corresponde a uma igualdade sintática, quer no caso da pré-condição quer no caso da pós-condição.

3. Caso *while*:

$$\frac{\dfrac{D_1}{\{\theta \wedge b\}\mathsf{C}\{\theta\}}}{\{\theta\}\texttt{while}\ b\ \{\mathsf{C}\}\{\theta \wedge \neg b\}}\ \text{While}$$

Queremos demonstrar que se $v \notin free(\texttt{while}\ b\ \{\mathsf{C}\})$ então $\vdash_{Sg} \{\exists v.\theta\}$ while $b\{\exists v.(\theta \wedge \neg b)\}$.

De sabermos que $v \notin free(\texttt{while}\ b\ \{\mathsf{C}\})$, temos em particular que $v \notin free(\mathsf{C})$. Daqui e da hipótese de indução, temos que existe uma derivação D'_1 em $\mathbf{S^g}$ com conclusão $\{\exists v.(\theta \wedge b)\}\mathsf{C}\{\exists v.\theta\}$.

Basta então tomar,

$$\dfrac{\dfrac{\dfrac{\dfrac{D'_1}{\{\exists v.(\theta \wedge b)\}\mathsf{C}\{\exists v.\theta\}}\,\text{Conseq.}}{\{(\exists v.\theta) \wedge b\}\mathsf{C}\{\exists v.\theta\}}}{\{\exists v.\theta\}\ \text{while}\ b\{\mathsf{C}\}\{\exists v.\theta \wedge \neg b\}}\,\text{While}}{\{\exists v.\theta\}\ \text{while}\ b\{\mathsf{C}\}\{\exists v.(\theta \wedge \neg b)\}}\,\text{Conseq.}$$

As condições laterais relativas às aplicações da consequência segue de $v \notin free(b)$ (que segue de $v \notin free(\texttt{while}\ b\ \{\mathsf{C}\})$).

4. Caso *if*:

$$\dfrac{\dfrac{D_1}{\{\phi \wedge b\}\mathsf{C}_t\{\psi\}} \quad \dfrac{D_2}{\{\phi \wedge \neg b\}\mathsf{C}_f\{\psi\}}}{\{\phi\}\texttt{if}\ b\ \texttt{then}\ \mathsf{C}_t\ \texttt{else}\ \mathsf{C}_f\{\psi\}}\,\text{If}$$

Queremos demonstrar que se $v \notin free(\texttt{if}\ b\ \texttt{then}\ \mathsf{C}_t\ \texttt{else}\ \mathsf{C}_f)$ então $\vdash_{Sg} \{\exists v.\phi\}\texttt{if}\ b\ \texttt{then}\ \mathsf{C}_t\ \texttt{else}\ \mathsf{C}_f\{\exists v.\psi\}$.

De sabermos que $v \notin free(\texttt{if}\ b\ \texttt{then}\ \mathsf{C}_t\ \texttt{else}\ \mathsf{C}_f)$, temos em particular que $v \notin free(\mathsf{C}_t)$ e ainda que $v \notin free(\mathsf{C}_f)$. Destes factos e das hipóteses de indução, temos que existem derivações D'_1 e D'_2 com conclusão $\{\exists v.(\phi \wedge b)\}\mathsf{C}_t\{\exists v.\psi\}$ e $\{\exists v.(\phi \wedge \neg b)\}\mathsf{C}_f\{\exists v.\psi\}$.

Basta então tomar,

$$\dfrac{\dfrac{\dfrac{D'_1}{\{\exists v.(\phi \wedge b)\}\mathsf{C}_t\{\exists v.\psi\}}\,\text{Conseq.}}{\{(\exists v.\phi) \wedge b\}\mathsf{C}_t\{\exists v.\psi\}} \quad \dfrac{\dfrac{D'_2}{\{\exists v.(\phi \wedge \neg b)\}\mathsf{C}_f\{\exists v.\psi\}}\,\text{Conseq.}}{\{(\exists v.\phi) \wedge \neg b\}\mathsf{C}_f\{\exists v.\psi\}}}{\{\exists v.\phi\}\texttt{if}\ b\ \texttt{then}\ \mathsf{C}_t\ \texttt{else}\ \mathsf{C}_f\{\exists v.\psi\}}\,\text{If}$$

No entanto, as condições laterais da inferência Conseq, seguem de $v \notin free(b)$ (uma consequência de $v \notin free(\texttt{if}\ b\ \texttt{then}\ \mathsf{C}_t\ \texttt{else}\ \mathsf{C}_f)$).

5. Caso consequência:

$$\dfrac{\models \phi' \Rightarrow \phi \quad \dfrac{D'_1}{\{\phi\}\mathsf{C}\{\psi\}} \quad \models \psi \Rightarrow \psi'}{\{\phi'\}\mathsf{C}\{\psi'\}}$$

Queremos demonstrar que se $v \notin free(\mathsf{C})$ então $\vdash_{Sg} \{\exists v.\phi'\}\mathsf{C}\{\exists v.\psi'\}$.

Como $v \notin free(\mathsf{C})$, da hipótese de indução temos que existe D'_1 derivação de $\{\exists v.\phi\}\mathsf{C}\{\exists v.\psi\}$.

Basta então tomar,

$$\dfrac{\dfrac{D'_1}{\{\exists v.\phi\}\mathsf{C}\{\exists v.\psi\}}}{\{\exists v.\phi'\}\mathsf{C}\{\exists v.\psi'\}}\ \text{Conseq.}$$

Falta agora demonstrar a validade das condições laterais da inferência da consequência, ou seja, temos que mostrar $\models \exists v.\phi' \Rightarrow \exists v.\phi$ e $\models \exists v.\psi \Rightarrow \exists v.\psi'$. Ora, suponhamos que $[\![\exists v.\phi']\!](s,h) \Leftrightarrow [\![\phi']\!]([s|v:a],h)$ para algum a. Daqui, e por sabermos que $\models \phi' \Rightarrow \phi$, temos, $[\![\phi]\!]([s|v:a],h)$ e portanto $[\![\exists x.\phi]\!](s,h)$.

A outra condição tem uma justificação análoga, que vem por também termos suposto que $\models \psi \Rightarrow \psi'$.

□

Anexo B

Protótipo do *VCGen*

B.1 Código

```
module Thesis where
import Data.Char

type Var=String

type VarI=String
type VarL=String

type Field=String

type Record = [(Field, Exp)]


data ExpInt = Const  Int
            | X      VarI
            | Menos  ExpInt
            | Soma   ExpInt     ExpInt
            | Mult   ExpInt     ExpInt
            | Div    ExpInt     ExpInt
            | Mod    ExpInt     ExpInt
          deriving Eq

instance Show ExpInt where
    show  = prettyPrint


data ExpBool= Top
            | Neg       ExpBool
            | E         ExpBool     ExpBool
            | Ou        ExpBool     ExpBool
            | Ig        ExpInt      ExpInt
            | Maior     ExpInt      ExpInt
            | Menor     ExpInt      ExpInt
            | MaiorIg   ExpInt      ExpInt
            | MenorIg   ExpInt      ExpInt
            | EqEnd     ExpLoc      ExpLoc
            deriving Eq

instance Show ExpBool where
    show = prettyPrintBool
```

```
data ExpLoc = Nil
            | P         VarL
            deriving Eq

instance Show ExpLoc where
    show = prettyPrintLoc

data Assert= N      Assert
           | And    Assert      Assert
           | Or     Assert      Assert
           | Impl   Assert      Assert
           | Eq     Assert      Assert
           | QQ     Var         Assert
           | Ex     Var         Assert
           | Emp
           | CSep   Assert      Assert
           | ImpSep Assert      Assert
           | Map    VarL        Record
           | Exp    ExpBool
           deriving Eq

instance Show Assert where
    show = prettyPrintAsser

data Exp = I ExpInt
         | L ExpLoc
         deriving (Eq,Show)



data Comando  = Skip
              | Atr         Var         Exp
              | Comp        Comando     Comando
              | If          ExpBool     Comando     Comando
              | While       ExpBool     Assert      Comando
              | Mutation    VarL        Field       Exp
              | New         VarL
              | Dispose     VarL
              | Lookup      VarI        VarL        Field
              | LookupF     VarL        VarL        Field
              deriving Eq

instance Show Comando where
    show = prettyPrintComando



vcgen :: Triplo -> [Assert]
vcgen (T phi c psi) =  norepetA [ Impl phi  (wp (c, psi)) ] (vc (c, psi))


data Triplo = T Assert Comando Assert
            deriving Show


vc::(Comando,Assert)->[Assert]
vc  (Skip,psi)              =[ ]
vc  (Atr p a, psi)          =[ ]
vc  (New p , psi)           =[ ]
vc  (Dispose a,psi)         =[ ]
vc  (Mutation a b c, psi)   =[ ]
vc  (Lookup v p f,psi)      =[ ]
vc  (LookupF v p f,psi)     =[ ]
vc  (Comp c1 c2, psi)       = norepetA (vc(c1,wp(c2,psi)))  (vc(c2,psi))
vc  (If b ct cf,psi)        = norepetA (vc(ct,psi)) (vc(cf,psi))
```

```
vc  (While b theta c,psi)   = norepetA (norepetA [Impl (And theta (Exp b)) (wp(c,↩
    psi)) ]  [Impl (And theta (N (Exp b))) psi])  (vc(c,theta))


wp::(Comando,Assert)->Assert
wp  (Skip,psi)              = psi
wp  (Atr v a, psi)          =substAssert psi v a
wp  (Comp c1 c2,psi)        =wp(c1, wp(c2,psi))
wp  (If t ct cf, psi)       =And (Impl (Exp t) ( wp(ct,psi) ) ) (Impl (Exp (Neg t)↩
    ) ( wp(cf,psi)) )
wp  (While b theta c,psi)   =theta
wp  (Dispose t ,psi)        =CSep psi (Map t [ ] )
wp  (New p,psi)             =let p1= (geraVar 0 "p '" (varsAssert psi ++ [p]) )
                             in (QQ p1 (ImpSep (Map  p1 [ ]) (substAssert psi p (L↩
                                 (P p1)) ) ) )
wp  (Mutation p f a ,psi)   =CSep (Map p [ ])(ImpSep (Map p [(f,a)] )(psi))
wp  (Lookup v p f,psi)      =let  p1= (geraVar 0 "v" (varsAssert psi ++[p]) )
                             in (Ex p1 (And (substAssert psi v (I (X p1)) ) (CSep ↩
                                 (Map p [(f,(I (X p1)) )]) (Exp Top))))
wp  (LookupF v p f,psi)     =let  p1= (geraVar 0 "v" (varsAssert psi ++[p]) )
                             in (Ex p1 (And (substAssert psi v (L (P p1)) ) (CSep ↩
                                 (Map p [(f,(L (P p1)) )]) (Exp Top))))



substExpInt::ExpInt->Var->ExpInt->ExpInt
substExpInt (Const a)    t e1 = (Const a)
substExpInt (X v)        t e1 = if v==t then e1
                                else (X v)
substExpInt (Menos e)    t e1 = (Menos (substExpInt e t e1))
substExpInt (Soma e1 e2) t e3 = (Soma (substExpInt e1 t e3) (substExpInt e2 t e3))
substExpInt (Mult e1 e2) t e3 = (Mult (substExpInt e1 t e3) (substExpInt e2 t e3))
substExpInt (Div e1 e2)  t e3 = (Div (substExpInt e1 t e3) (substExpInt e2 t e3))
substExpInt (Mod e1 e2)  t e3 = (Mod (substExpInt e1 t e3) (substExpInt e2 t e3))


subsRecord::Record->Var->Exp->Record
subsRecord [ ]        v e1 = [ ]
subsRecord ((f, e):t) v e1 = ((f,(substExp e v e1)): subsRecord t v e1)


substBool::ExpBool->Var->Exp->ExpBool
substBool (Top)              p v     = Top
substBool (Neg b)            p v     = (Neg (substBool b p v))
substBool (E b1 b2)          p v     = (E (substBool b1 p v) (substBool b2 p v))
substBool (Ou b1 b2)         p v     = (Ou (substBool b1 p v) (substBool b2 p v))
substBool (Ig e1 e2)         p (I v) = (Ig (substExpInt e1 p v) (substExpInt e2 p ↩
    v))
substBool (Ig e1 e2)         p (L v) = (Ig e1 e2)
substBool (Maior e1 e2)      p (I v) = (Maior (substExpInt e1 p v) (substExpInt e2↩
     p v))
substBool (Maior e1 e2)      p (L v) = (Maior e1 e2)
substBool (Menor e1 e2)      p (I v) = (Menor (substExpInt e1 p v) (substExpInt e2↩
     p v))
substBool (Menor e1 e2)      p (L v) = (Menor e1 e2 )
substBool (MaiorIg e1 e2)    p (I v) = (MaiorIg (substExpInt e1 p v) (substExpInt ↩
    e2 p v))
substBool (MaiorIg e1 e2)    p (L v) = (MaiorIg e1 e2)
substBool (MenorIg e1 e2)    p (I v) = (MenorIg (substExpInt e1 p v) (substExpInt ↩
    e2 p v))
substBool (MenorIg e1 e2)    p (L v) = (MenorIg e1 e2)
substBool (EqEnd l1 l2)      p (L v) = (EqEnd (substExpLoc l1 p v) (substExpLoc l2↩
     p v))
substBool (EqEnd l1 l2)      p (I v) = (EqEnd l1 l2)
```

```
substAssert::Assert->Var->Exp->Assert
substAssert (N phi)          p v = (N (substAssert phi p v))
substAssert (And phi psi)    p v = (And (substAssert phi p v) (substAssert psi p v)
    )
substAssert (Or phi psi)     p v = (Or (substAssert phi p v) (substAssert psi p v))
substAssert (Impl phi psi)   p v = (Impl (substAssert phi p v) (substAssert psi p v
    ))
substAssert (Eq phi psi)     p v = (Eq (substAssert phi p v) (substAssert psi p v))
substAssert (QQ v1 phi )     p v = if v1==p \& \& ((existeVar p (varsAssert (QQ v1
    phi))) || existeVars (varsExp v) (varsLigadas (QQ v1 phi))) then
                                     let k=varsAssert(phi) ++ varsExp(v) ++ [p]
                                         fresh=geraVar 0 p k
                                     in substAssert (renomeiaAssert (QQ v1 phi) v1
                                         fresh) p v
                                    else (QQ v1 (substAssert phi p v))
substAssert (Ex v1 phi)      p v = if v1==p \&\& (existeVar p (varsAssert (Ex v1
    phi))) || existeVars (varsExp v) (varsLigadas (Ex v1 phi)) then
                                     let k=varsAssert(phi) ++ varsExp(v) ++ [p]
                                         fresh=geraVar 0 p k
                                     in substAssert (renomeiaAssert (Ex v1 phi) v1
                                         fresh) p v
                                   else (Ex v1 (substAssert phi p v))
substAssert (Emp)            p v = Emp
substAssert (CSep phi psi)   p v = (CSep (substAssert phi p v) (substAssert psi p v
    ))
substAssert (ImpSep phi psi)p v = (ImpSep (substAssert phi p v) (substAssert psi p
     v))
substAssert (Map l r)        p v = (Map l (subsRecord r p v))
substAssert (Exp b)          p v = (Exp (substBool b p v))


substExpLoc::ExpLoc->Var->ExpLoc->ExpLoc
substExpLoc Nil   p l1   = Nil
substExpLoc (P l) p l1   = if l==p then l1
                           else (P l)


substExp::Exp->Var->Exp->Exp
substExp (I e) l (I e1) = (I (substExpInt e l e1))
substExp (I e) l (L e1) = (I e)
substExp (L e) l (L e1) = (L (substExpLoc e l e1))
substExp (L e) l (I e1) = (L e)


tese1::Assert
tese1= And (Exp (Ig (X "x")(Const 2)) ) (Exp (Ig (X "y") (Const 1)))

tese2::Assert
tese2= And (Exp (Ig (X "x")(Const 1)) ) (Exp (Ig (X "y") (Const 2)))

tesecom1::Comando
tesecom1 = Comp (New "p")(Comp (Mutation "p" "Valor"  (I (X "x"))) (Comp (Atr "x"
    (I (X "y"))) (Comp (Lookup  "y" "p" "valor") (Dispose "p"))))


varsComando::Comando->[Var]
varsComando Skip                =[ ]
varsComando (Atr v (I e))       =[v] ++ vars(e)
varsComando (Atr v (L l))       =[v] ++ varsLoc(l)
varsComando (If b c1 c2)        =(varsBool b) ++ (varsComando c1) ++ (varsComando
     c2)
varsComando (While b phi c)     =(varsBool b) ++ (varsAssert phi) ++ (varsComando
     c)
varsComando (Mutation p f (I e)) = [p] ++ vars(e)
varsComando (Mutation p f (L l)) = [p] ++ varsLoc(l)
varsComando (New p)             = [p]
varsComando (Dispose p)         = [p]
```

```
varsComando (Lookup v p f)       = [v , p]
varsComando (LookupF v p f)      = [v , p]

prettyPrintRecord::Record -> String
prettyPrintRecord [ ]            = "  "
prettyPrintRecord ((f, (I e)):t) = "("++ f ++ "," ++ prettyPrint e ++ ")" ++ ↩
    prettyPrintRecord t
prettyPrintRecord ((f, (L l)):t) = "("++ f ++ "," ++ prettyPrintLoc l ++ ")" ++ ↩
    prettyPrintRecord t



prettyPrint::ExpInt->String
prettyPrint(Const a)   = show(a)
prettyPrint(X p)       = p
prettyPrint(Menos a)   = "-"++ prettyPrint a
prettyPrint(Soma a b)  = prettyPrint a ++ " + " ++ prettyPrint b
prettyPrint(Mult a b)  = prettyPrint a ++ " * " ++ prettyPrint b
prettyPrint(Div a b)   = prettyPrint a ++ " / " ++ prettyPrint b
prettyPrint(Mod a b)   = prettyPrint a ++ " Mod " ++ prettyPrint b

prettyPrintBool::ExpBool->String
prettyPrintBool Top             ="True"
prettyPrintBool (Neg a)         ="(" ++"~ " ++ prettyPrintBool a ++ ")"
prettyPrintBool (E a b)         ="(" ++        prettyPrintBool a ++ " \&\& "  ++ ↩
    prettyPrintBool b ++")"
prettyPrintBool (Ou a b)        ="(" ++        prettyPrintBool a ++ " || "  ++ ↩
    prettyPrintBool b ++")"
prettyPrintBool (Ig a b)        ="(" ++        prettyPrint     a ++ " = "   ++ ↩
    prettyPrint b     ++")"
prettyPrintBool (Maior a b )    ="(" ++        prettyPrint     a ++ " > "   ++ ↩
    prettyPrint b     ++")"
prettyPrintBool (Menor a b )    ="(" ++        prettyPrint     a ++ " < "   ++ ↩
    prettyPrint b     ++")"
prettyPrintBool (MaiorIg a b )  ="(" ++        prettyPrint     a ++ " = > " ++ ↩
    prettyPrint b     ++")"
prettyPrintBool (MenorIg a b )  ="(" ++        prettyPrint     a ++ " < = " ++ ↩
    prettyPrint b     ++")"
prettyPrintBool (EqEnd p1 p2)   ="(" ++        prettyPrintLoc  p1++ "="     ++ ↩
    prettyPrintLoc p2 ++ ")"


prettyPrintComando::Comando->String
prettyPrintComando Skip             = "Skip"
prettyPrintComando (Comp c1 c2)     = prettyPrintComando c1 ++ ";" ++ ↩
    prettyPrintComando c2
prettyPrintComando (If b c1 c2)     = "If" ++ prettyPrintBool b ++ " then " ++ ↩
    prettyPrintComando c1 ++ "else" ++ prettyPrintComando c2
prettyPrintComando (While b theta c)= "While" ++ prettyPrintBool b ++ ↩
    prettyPrintAsser theta ++ prettyPrintComando c
prettyPrintComando (New p)          = "New (" ++ p ++ ")"
prettyPrintComando (Dispose e)      = "Dispose " ++ e
prettyPrintComando (Lookup a b c)   = a ++ ":=" ++ b ++"->" ++ c
prettyPrintComando (LookupF a b c)  = a ++ ":=" ++ b ++"->" ++ c

prettyPrintAsser::Assert->String
prettyPrintAsser (N a)       = "("         ++"~" ++ prettyPrintAsser a ++")"
prettyPrintAsser (And a b)   = "("         ++ prettyPrintAsser a ++ " \&\& " ++ ↩
    prettyPrintAsser  b ++")"
prettyPrintAsser (Or a b)    = "("         ++ prettyPrintAsser a ++ " || " ++ ↩
    prettyPrintAsser  b ++")"
prettyPrintAsser (Impl a b)  = "("         ++ prettyPrintAsser a ++ " => " ++ ↩
    prettyPrintAsser  b ++")"
prettyPrintAsser (Eq a b)    = "("         ++ prettyPrintAsser a ++ " <=> "++ ↩
    prettyPrintAsser  b ++")"
prettyPrintAsser (QQ a b)    = "(qualquer" ++ a ++ ". "                    ++ ↩
    prettyPrintAsser  b ++")"
```

```
prettyPrintAsser (Ex a b)    = "(exists"   ++ a ++ ". "                    ++ ↩
    prettyPrintAsser  b ++")"
prettyPrintAsser (Emp)       = "EMP "
prettyPrintAsser (CSep a b)  = "("         ++ prettyPrintAsser a ++ " * "  ++ ↩
    prettyPrintAsser  b ++")"
prettyPrintAsser (ImpSep a b)= "("         ++ prettyPrintAsser a ++ " -* " ++ ↩
    prettyPrintAsser  b ++")"
prettyPrintAsser (Map a b)   = "("         ++ a ++ " |-> "                ++ ↩
    prettyPrintRecord b ++")"
prettyPrintAsser (Exp b )    = prettyPrintBool b

prettyPrintLoc::ExpLoc -> String
prettyPrintLoc Nil   = "Nil"
prettyPrintLoc (P p) = p



norepet::[Assert]->Assert-> [Assert]
norepet [ ]     phi = [phi]
norepet (h:t)   phi =if h==phi then (h:t)
                     else h:(norepet t phi)

norepetA::[Assert]->[Assert]->[Assert]
norepetA  a [ ]   = a
norepetA  a (h:t) = norepetA (norepet a h) t


geraVar::Int->Var->[Var]->Var
geraVar i p [ ]     = p ++ show(i)
geraVar i p (a:h)   = if  (p ++ show(i)) ==a then geraVar (i+1)  p (a:h)
                      else (geraVar i p h)

existeVar::Var->[Var]->Bool
existeVar p [ ]   = False
existeVar p (h:t) = if p==h then True
                    else existeVar p t

existeVars::[Var]->[Var]->Bool
existeVars [ ]      [ ] = False
existeVars h        [ ] = True
existeVars [ ]      h   = False
existeVars (h:t)    h1  = (existeVar h h1) \&\& existeVars t h1


subst::ExpInt->Var->ExpInt->ExpInt
subst (Const a)     p e = (Const a)
subst (X p1)        p e = if p1==p then e
                          else (X p1)
subst (Menos e1)    p e = Menos (subst e1 p e)
subst (Soma e1 e2)  p e = Soma  (subst e1 p e) (subst e2 p e)
subst (Mult e1 e2)  p e = Mult  (subst e1 p e) (subst e2 p e)
subst (Div e1 e2)   p e = Div   (subst e1 p e) (subst e2 p e)
subst (Mod e1 e2)   p e = Mod   (subst e1 p e) (subst e2 p e)


varsRecord::Record->[Var]
varsRecord [ ]       = [ ]
varsRecord ((f,e):t) = (varsExp e) ++ varsRecord t

renomeiaRecord::Record->Var->Var->Record
renomeiaRecord [ ] p1 p2       = [ ]
renomeiaRecord ((f,e):t) p1 p2 = (f, (renomeiaExp e p1 p2)) : (renomeiaRecord t p1↩
     p2)

renomeiaExpInt::ExpInt->Var->Var->ExpInt
renomeiaExpInt (Const a)    p1 p2 = (Const a)
renomeiaExpInt (X v)        p1 p2 = if v==p1 then (X p2)
```

```
                                    else (X v)
renomeiaExpInt (Menos e)    p1 p2 = (Menos (renomeiaExpInt e p1 p2))
renomeiaExpInt (Soma e1 e2) p1 p2 = (Soma (renomeiaExpInt e1 p1 p2)     (↩
    renomeiaExpInt e2 p1 p2))
renomeiaExpInt (Mult e1 e2) p1 p2 = (Mult (renomeiaExpInt e1 p1 p2)     (↩
    renomeiaExpInt e2 p1 p2))
renomeiaExpInt (Div  e1 e2) p1 p2 = (Div (renomeiaExpInt e1 p1 p2)      (↩
    renomeiaExpInt e2 p1 p2))
renomeiaExpInt (Mod  e1 e2) p1 p2 = (Mod (renomeiaExpInt e1 p1 p2)      (↩
    renomeiaExpInt e2 p2 p2))

vars::ExpInt->[Var]
vars (Const a)      = [ ]
vars (X p)          =[p]
vars (Menos e)      = vars e
vars (Soma e1 e2)   = vars(e1) ++ vars(e2)
vars (Mult e1 e2)   = vars(e1) ++ vars(e2)
vars (Div e1 e2)    = vars(e1) ++ vars(e2)
vars (Mod e1 e2)    = vars(e1) ++ vars(e2)

renomeiaBool::ExpBool->Var->Var->ExpBool
renomeiaBool (Top)           p1 p2 = (Top)
renomeiaBool (Neg b)         p1 p2 = (Neg (renomeiaBool b p1 p2))
renomeiaBool (E b1 b2)       p1 p2 = (E (renomeiaBool b1 p1 p2) (renomeiaBool b2 ↩
    p1 p2))
renomeiaBool (Ou b1 b2)      p1 p2 = (Ou (renomeiaBool b1 p1 p2) (renomeiaBool b2 ↩
    p1 p2))
renomeiaBool (Ig b1 b2)      p1 p2 = (Ig (renomeiaExpInt b1 p1 p2) (renomeiaExpInt↩
     b2 p1 p2))
renomeiaBool (Maior e1 e2)   p1 p2 = (Maior (renomeiaExpInt e1 p1 p2) (↩
    renomeiaExpInt e2 p1 p2))
renomeiaBool (Menor e1 e2)   p1 p2 = (Menor (renomeiaExpInt e1 p1 p2) (↩
    renomeiaExpInt e2 p1 p2))
renomeiaBool (MaiorIg e1 e2) p1 p2 = (MaiorIg (renomeiaExpInt e1 p1 p2) (↩
    renomeiaExpInt e2 p1 p2))
renomeiaBool (MenorIg e1 e2) p1 p2 = (MenorIg (renomeiaExpInt e1 p1 p2) (↩
    renomeiaExpInt e2 p1 p2))
renomeiaBool (EqEnd l1 l2)   p1 p2 = (EqEnd (renomeiaExpLoc l1 p1 p2) (↩
    renomeiaExpLoc l2 p1 p2))


renomeiaExp::Exp->Var->Var->Exp
renomeiaExp (I e) p p1 = (I (renomeiaExpInt e p p1))
renomeiaExp (L l) p p1 = (L (renomeiaExpLoc l p p1))


varsBool::ExpBool->[Var]
varsBool (Top)           = [ ]
varsBool (Neg     b)     = varsBool b
varsBool (E       b1 b2) = (varsBool b1) ++ (varsBool b2)
varsBool (Ou      b1 b2) = (varsBool b1) ++ (varsBool b2)
varsBool (Ig      e1 e2) = (vars e1)     ++ (vars e2)
varsBool (Maior   e1 e2) = (vars e1)     ++ (vars e2)
varsBool (Menor   e1 e2) = (vars e1)     ++ (vars e2)
varsBool (MaiorIg e1 e2) = (vars e1)     ++ (vars e2)
varsBool (MenorIg e1 e2) = (vars e1)     ++ (vars e2)
varsBool (EqEnd   l1 l2) = (varsLoc l1)  ++ (varsLoc l2)


        --Var 1 variavel a ser substituida
        --Var 2 variavel que vai aparecer
renomeiaAssert::Assert->Var->Var->Assert
renomeiaAssert (N phi)           p1 p2 = (N (renomeiaAssert phi p1 p2))
renomeiaAssert (And phi psi)     p1 p2 = (And (renomeiaAssert phi p1 p2) (↩
    renomeiaAssert psi p1 p2))
renomeiaAssert (Or phi psi)      p1 p2 = (Or (renomeiaAssert phi p1 p2) (↩
    renomeiaAssert psi p1 p2))
```

```
renomeiaAssert (Impl phi psi)    p1 p2 = (Impl (renomeiaAssert phi p1 p2) (↩
    renomeiaAssert psi p1 p2))
renomeiaAssert (Eq phi psi)      p1 p2 = (Eq (renomeiaAssert phi p1 p2) (↩
    renomeiaAssert psi p1 p2))
renomeiaAssert (QQ v phi)        p1 p2 = if v==p1 then (QQ p2 (renomeiaAssert phi ↩
    p1 p2))
                                         else (QQ v (renomeiaAssert phi p1 p2))
renomeiaAssert (Ex v phi)        p1 p2 = if v==p1 then (Ex p2 (renomeiaAssert phi ↩
    p1 p2))
                                         else (Ex v (renomeiaAssert phi p1 p2))
renomeiaAssert (Emp)             p1 p2 = Emp
renomeiaAssert (CSep phi psi)    p1 p2 = (CSep  (renomeiaAssert phi p1 p2) (↩
    renomeiaAssert psi p1 p2))
renomeiaAssert (ImpSep phi psi)  p1 p2 =(ImpSep (renomeiaAssert phi p1 p2) (↩
    renomeiaAssert psi p1 p2))
renomeiaAssert (Map l r)         p1 p2 = if l==p1 then (Map p2 (renomeiaRecord r ↩
    p1 p2))
                                         else (Map l (renomeiaRecord r p1 p2))
renomeiaAssert (Exp b)           p1 p2 = (Exp (renomeiaBool b p1 p2))


varsLigadas::Assert->[Var]
varsLigadas (N       phi)     = (varsLigadas phi)
varsLigadas (And     phi psi) = (varsLigadas phi) ++ (varsLigadas psi)
varsLigadas (Or      phi psi) = (varsLigadas phi) ++ (varsLigadas psi)
varsLigadas (Impl    phi psi) = (varsLigadas phi) ++ (varsLigadas psi)
varsLigadas (Eq      phi psi) = (varsLigadas phi) ++ (varsLigadas psi)
varsLigadas (QQ v    phi)     = [v]               ++ (varsLigadas phi)
varsLigadas (Ex v    phi)     = [v]               ++ (varsLigadas phi)
varsLigadas (Emp)             = [ ]
varsLigadas (CSep    phi psi) = (varsLigadas phi) ++ (varsLigadas psi)
varsLigadas (ImpSep  phi psi) = (varsLigadas phi) ++ (varsLigadas psi)
varsLigadas (Map     l r)     = [ ]
varsLigadas (Exp     b)       = [ ]

varsAssert::Assert->[Var]
varsAssert (N          phi)     = varsAssert phi
varsAssert (And        phi psi) = (varsAssert phi) ++ (varsAssert psi)
varsAssert (Or         phi psi) = (varsAssert phi) ++ (varsAssert psi)
varsAssert (Impl       phi psi) = (varsAssert phi) ++ (varsAssert psi)
varsAssert (Eq         phi psi) = (varsAssert phi) ++ (varsAssert psi)
varsAssert (QQ         v   phi) = [v]              ++ (varsAssert phi)
varsAssert (Ex         v   phi) = [v]              ++ (varsAssert phi)
varsAssert (Emp)                = [ ]
varsAssert (CSep       phi psi) = (varsAssert phi) ++ (varsAssert psi)
varsAssert (ImpSep     phi psi) = (varsAssert phi) ++ (varsAssert psi)
varsAssert (Map        v   r)   = [v]              ++ (varsRecord r)
varsAssert (Exp        b)       = varsBool b

varsLoc::ExpLoc->[Var]
varsLoc (Nil) =[ ]
varsLoc (P l) =[l]

renomeiaExpLoc::ExpLoc->Var->Var->ExpLoc
renomeiaExpLoc (Nil) p1 p2 = (Nil)
renomeiaExpLoc (P l) p1 p2 = if l==p1 then (P p2)
                             else (P l)


varsExp::Exp->[Var]
varsExp (I e) = vars(e)
varsExp (L l) =varsLoc(l)
```